Fundamentals of Java

Lesson 4: Introduction to Control Statements

Text by: Lambert and Osborne
Slides by: Cestroni
Additions and Modifications by:
Mr. Dave Clausen

Updated for Java 5 (version 1.5)

Lesson 4: Introduction to Control Statements

Objectives:

- Use the increment and decrement operators.
- Use standard math methods.
- Use if and if-else statements to make choices.
- Use while and for loops to repeat a process.
- Construct appropriate conditions for control statements using relational operators.
- Detect and correct common errors involving loops.

Lesson 4: Introduction to Control Statements

Vocabulary:

- control statements
- counter
- count-controlled loop
- flowchart
- infinite loop
- iteration
- off-by-one error
- overloading
- random walk
- sentinel
- task-controlled loop

4.1 Additional Operators

Extended Assignment Operators

- The assignment operator can be combined with the arithmetic and concatenation operators to provide extended assignment operators. For example

```
int a = 17;
String s = "hi";
a += 3;                 // Equivalent to  a =  a + 3;
a -= 3;                 // Equivalent to  a =  a – 3;
a *= 3;                 // Equivalent to  a =  a * 3;
a /= 3;                 // Equivalent to  a =  a / 3;
a %= 3;                 // Equivalent to  a =  a % 3;
s += " there";    // Equivalent to  s =  s + " there";
```

4.1 Additional Operators

- Extended assignment operators can have the following format.

  variable op= expression;

  which is equivalent to

  variable = variable op expression;

- Note that there is no space between op and =.
- The extended assignment operators and the standard assignment have the same precedence.

4.1 Additional Operators

Increment and Decrement

- Java includes increment (++) and decrement (--) operators that increase or decrease a variables value by one:

```
int m = 7;
double x = 6.4;
m++;        // Equivalent to m = m + 1;
x--;        // Equivalent to x = x – 1.0;
```

- The precedence of the increment and decrement operators is the same as unary plus, unary minus, and cast.

4.2 Standard Classes and Methods

The Math Class

- Notice that two methods in the table are called abs. They are distinguished from each other by the fact that one takes an integer and the other takes a double parameter.

- Using the same name for two different methods is called ***overloading***

4.2 Standard Classes and Methods

Seven of the methods in the Math Class

METHOD	WHAT IT DOES
`static int abs(int x)`	Returns the absolute value of an integer `x`
`static double abs(double x)`	Returns the absolute value of a double `x`
`static double pow(double base, double exponent)`	Returns the `base` raised to the `exponent`
`static long round(double x)`	Returns `x` rounded to the nearest whole number (Note: Returned value must be cast to an `int` before assignment to an `int` variable.)
`static int max(int a, int b)`	Returns the greater of `a` and `b`
`static int min(int a, int b)`	Returns the lesser of `a` and `b`
`static double sqrt(double x)`	Returns the square root of `x`

abs() and Method Overloading

- The two `abs()` methods are **overloaded**.
- **Overloaded**: Multiple methods in the same class with the same name
- Using `sqrt()` method example:

```
// Given the area of a circle, compute its radius.
// Use the formula a = pr², where a is the area and r is the radius.

double area = 10.0, radius;
radius = Math.sqrt(area / Math.PI);
```

4.2 Standard Classes and Methods

The sqrt Method

- This code segment illustrates the use of the sqrt method:

```
// Given the area of a circle, compute its radius
// Use the formula a = πr2 , where a is the area and
// r is the radius
double area = 10.0, radius;
radius = Math.sqrt (area / Math.PI);
```

- Messages are usually sent to objects; however, if a method's signature is labeled static, the message instead is sent to the method's class.

4.2 Standard Classes and Methods

The sqrt Method

- Thus, to invoke the sqrt method, we send the sqrt message to the Math class.
- In addition to methods, the Math class includes good approximations to several important constants.
- Math.PI is an approximation for π accurate to about 17 decimal places.

Math class methods examples

```
int m;
double x;

m = Math.abs(-7);                 // m equals 7
x = Math.abs(-7.5);               // x equals 7.5

x = Math.pow(3.0, 2.0);           // x equals 3.0^2.0 equals 9.0
x = Math.pow(16.0, 0.25);         // x equals 16.0^0.25 equals 2.0

m = Math.max(20, 40);             // m equals 40
m = Math.min(20, 40);             // m equals 20
m = (int) Math.round(3.14);       // m equals 3
m = (int) Math.round(3.5);        // m equals 4
```

4.2 Standard Classes and Methods

The Random Class

- A ***random number generator*** returns numbers chosen at random from a predesignated interval.
- Java's random number generator is implemented in the Random class and utilizes the methods nextInt and nextDouble as described in table 4-2.

METHOD	WHAT IT DOES
`int nextInt(int n)`	Returns an integer chosen at random from among 0, 1, 2, ..., *n* – 1
`double nextDouble()`	Returns a double chosen at random between 0.0 and 1.0, inclusive.

- A program that uses the Random class first must import java.util.Random.

Random Class Example

```
import java.util.Scanner;
import java.util.Random;

//Generate an integer between 0 and user's upper limit
Scanner reader = new Scanner(System.in);
Random generator = new Random ();

System.out.print("Enter the upper limit for a random int: ");
upperLimit = reader.nextInt();
System.out.print(generator.nextInt(upperLimit ));
```

Math.random()

- The College Board tests on Math.random() instead of importing the Random class and using nextInt(), etc.
- Math.Random generates a decimal within the range: 0.0 <= random <1.0
- To generate integers, use this formula:

randomInt= (int) (Math.random()*(HIGH - LOW + 1))+ LOW;

Where HIGH & LOW are constants representing the largest and smallest integers you wish to have generated.

Control Structures

- Corrado Bohm & Guiseppe Jacopini
  - 1964 Structure Theorem

    proved that any program logic, regardless of the complexity, can be expressed using the control structures of sequencing, selection, and repetition.

Control Structures 2

- A. Sequence
  - Instructions executed in order 1st , 2nd , 3rd, etc.
- B. Selection
  - Branching, Conditionals
    - if, if else
    - Switch statements (not part of APCS Java Subset)

Control Structures 3

- C. Repetition (Iteration)
  - Indefinite Loops
    - while loop
      - (condition checked at beginning)
    - do…while (not part of APCS Java Subset)
      - (condition checked at the end of the loop)
  - Definite Loops
    - for loop
- D. Recursion

Pseudocode

```
herd the cows from the field into the west paddock;
while (there are any cows left in the west paddock){
    fetch a cow from the west paddock;
    tie her in the stall;
    if (she is red){
        milk her into the red bucket;
        pour the milk into red bottles;
   }else{
        milk her into the black bucket;
        pour the milk into black bottles;
   }
   put her into the east paddock;
}
herd the cows from the east paddock back into the field;
clean the buckets;
```

4.4 Control Statements

- While and if-else are called ***control statements***.

```
while (some condition)
{
   do stuff;
}
```

Means do the stuff repeatedly as long as the condition holds true

Means if some condition
is true,
do stuff 1,
and if it is false,
do stuff 2.

```
if (some condition)
{
   do stuff 1;
}
else
{
   do stuff 2;
}
```

4.5 The if and if-else Statements

Principal Forms

- In Java, the if and if-else statements allow for the conditional execution of statements.

```
if (condition){
   statement;          //Execute these statements if the
   statement;          //condition is true.
}

if (condition){
   statement;          //Execute these statements if the
   statement;          //condition is true.
}else{
   statement;          //Execute these statements if the
   statement;          //condition is false.
}
```

4.5 The if and if-else Statements

- The indicated semicolons and braces are required
- Braces always occur in pairs
- There is no semicolon immediately following a closing brace.

The `if` and `if-else` Statements (cont.)

- Additional forms:

```
if (condition)
    statement;

if (condition)
    statement;
else
    statement;
```

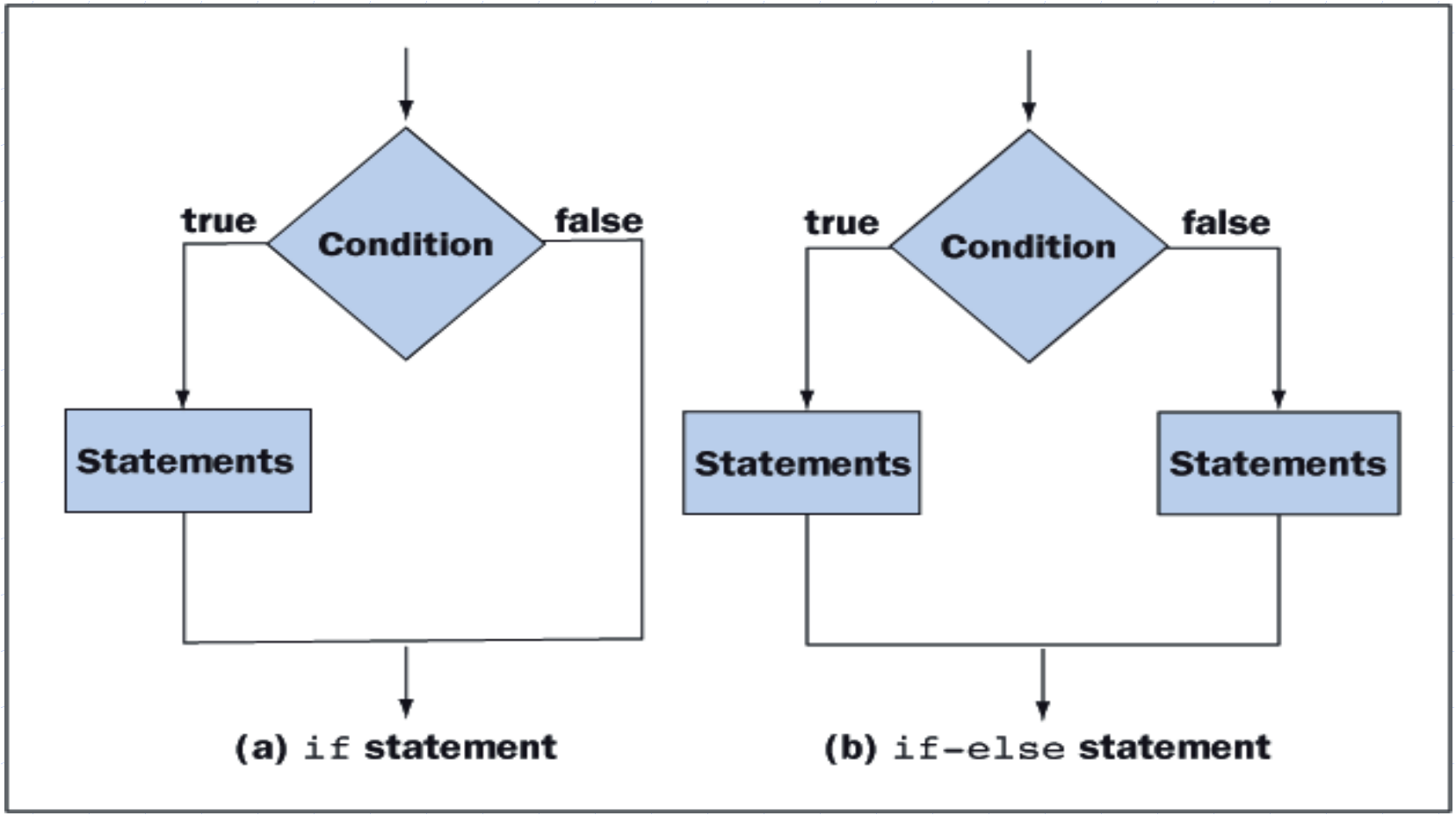
```
if (condition){
    statement;
        ...
    statement;
}else
    statement;

if (condition)
    statement;
else{
    statement;
        ...
    statement;
}
```

The `if` and `if-else` Statements (cont.)

- Better to over-use braces than under-use them
  - Can help to eliminate logic errors
- Condition of an `if` statement must be a **Boolean expression**
  - Evaluates to `true` or `false`
- A **flowchart** can be used to illustrate the behavior of `if-else` statements.

4.5 The if and if-else Statements

Figure 4-1 shows a diagram called a ***flowchart*** that illustrates the behavior of if and if-else statements.



4.5 The if and if-else Statements

Examples:

```
// Increase a salesman's commission by 10% if his sales are over $5000
if (sales > 5000)
   commission *= 1.1;

// Pay a worker $14.5 per hour plus time and a half for overtime
pay = hoursWorked * 14.5;
if (hoursWorked > 40){
   overtime = hoursWorked - 40;
   pay += overtime * 21.75;
}

// Let c equal the larger of a and b
if (a > b)
   c = a;
else
   c = b;
```

4.5 The if and if-else Statements

Relational Operators

Table 4-3 shows the complete list of relational operators available for use in Java.

OPERATOR	WHAT IT MEANS
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

4.5 The if and if-else Statements

- The double equal signs (==) distinguish the equal to (equality comparison) operator from the assignment operator (=).
- Side effects are caused if we confuse the two operators.
- In the not equal to operator, the exclamation mark (!) is read as not.

Checking for Valid Input

Example 4.1: Computes the area of a circle if the radius >= 0 or otherwise displays an error message. Error trapping with if else statements work best in GUIs, for Terminal IO use while.

```
import java.util.Scanner;

public class CircleArea{

   public static void main(String[] args){
      Scanner reader = new Scanner(System.in);
      System.out.print("Enter the radius: ");
      double radius = reader.nextDouble();
      if (radius < 0)
         System.out.println("Error: Radius must be >= 0");
      else{
         double area = Math.PI * Math.pow(radius, 2);
         System.out.println("The area is " + area);
      }
   }
}
```

Avoid Sequential Selection

- 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 nested selection (see Ch. 6)
if (condition2)
     action2
if (condition3)
     action3
```

Compound Boolean Expressions

- Logical Operators
  - And && (two ampersands) Conjunction
  - Or || (two pipe symbols) Disjunction
  - Not ! (one exclamation point) Negation
- Use parentheses for each simple expression and the logical operator between the two parenthetical expressions.
  - i.e. ((grade >= 80) && (grade < 90))

Truth Tables for AND / OR

Expression1 (E1)	Expression2 (E2)	E1&&E2	E1||E2
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Truth Table for NOT
Order of Priority for Booleans

Expression	Not E
(E)	!E
true	false
false	true

- Order Of Priority in Boolean Expressions
  - 1. !
  - 2. &&
  - 3. ||

Order of Operations including Booleans

1. ( )
2. !
3. *, /, %
4. +, -
5. <, <=, >, >=, = =, !=
6. &&
7. ||

Complements

Operation	Complement (equivalent)
! <	>=
! <=	>
! >	<=
! >=	<

De Morgan's Laws

! (A && B) is the same as:

Not (true AND true)

Not (true)

false

!A || !B

Not(true) OR Not(true)

false OR false

false

!(A || B) is the same as:

Not(true OR true)

Not (true)

false

!A && !B

Not(true) AND Not(true)

false AND false

false

Distributive Property of Logic

- A || (B && C)

  (A || B) && (A || C)

// Parenthesis are required due to logical precedence.

- A && (B || C)

  (A && B) || (A && C)

//Parenthesis not required for this example, but suggested.

Boolean Algebra

Law	&& major form	|| major form
Absorption	A && (A || B) ≡ A	A || (A && B) ≡ A
Associativity	A && (B && C) ≡ (A && B) && C	A || (B || C) ≡ (A || B) || C
Commutivity	A && B ≡ B && A	A || B ≡ B || A
Complementarity	A && !A ≡ false	A || !A ≡ true
DeMorgan's Laws	!(A && B) ≡ !A || !B	!(A || B) ≡ !A && !B
Distributivity	A && (B || C) ≡ (A && B)||(A && C)	A||(B && C) ≡ (A || B) && (A || C)
Idempotence	A && A ≡ A	A || A ≡ A
Identity	A && true ≡ A	A || false ≡ A
Universal Bounds	A && false ≡ false	A || true ≡ true

Absorption Law Proof

&& Major Form
A && (A || B) ≡ A

A && (A || B)
|| identity gives us
(A || false) && (A || B)

Distributive Prop ("factor" out A)
A || (false && B)

Universal Bounds for false && B
A || false

|| identity
A

|| Major Form
A || (A && B) ≡ A

A || (A && B)
&& identity gives us
(A && true) || (A && B)

Distributive Prop ("factor" out A)
A && (true || B)

Universal Bounds for true || B
A && true

&& identity
A

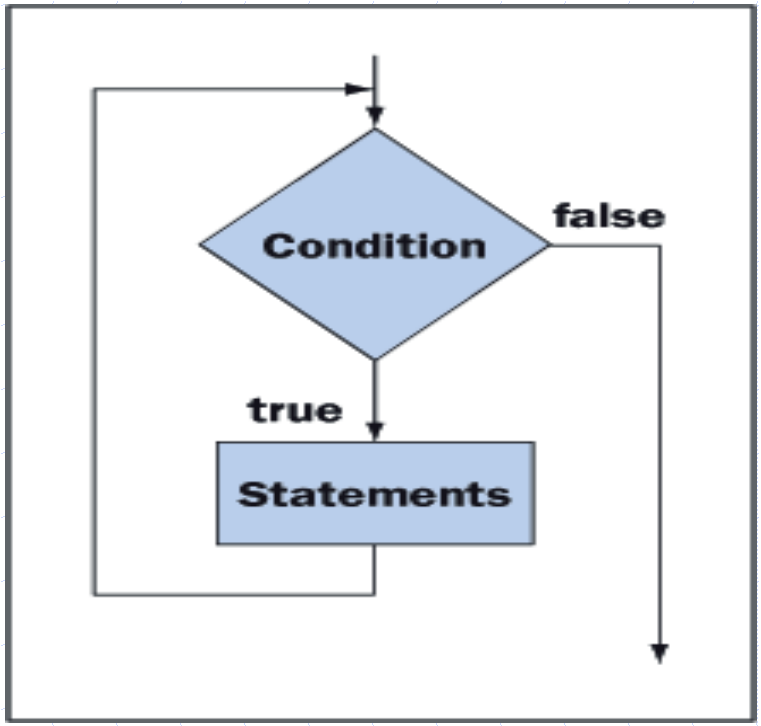
4.6 The while Statement

- The while statement provides a looping mechanism that executes statements repeatedly for as long as some condition remains true.

```
while (condition)       //loop test
    statement;          //one statement inside the loop body

while (condition)       //loop test
{
    statement;                  //many statements
    statement;                  //inside the
    ...                         //loop body
}
```

4.6 The while Statement

- If the condition is false from the outset, the statement or statements inside the loop never execute. Figure 4-2 uses a flowchart to illustrate the behavior of a while statement.



4.6 The while Statement

Compute 1+2+…+100 (**Count-controlled Loops**)

- The following code computes and displays the sum of the integers between 1 and 100, inclusive:

```
// Compute 1 + 2 + ... + 100
int sum = 0, counter = 1;  //Initialize sum and counter before loop
while (counter <= 100)
{
      sum += counter;      //point p (we refer to this location in Table 4-4)
       counter ++;         // point q (we refer to this location in table 4-4)
}
System.out.println  (sum);
```

4.6 The while Statement: count controlled loop

- The variable *counter* acts as a counter that controls how many times the loop executes.
- The *counter* starts at 1.
- Each time around the loop, it is compared to 100 and incremented by 1.
- The code inside the loop is executed exactly 100 times, and each time through the loop , sum is incremented by increasing values of *counter* .
- The variable *counter* is called the ***counter.***

4.6 The while Statement

Tracing the Variables

- To understand the loop fully, we must analyze the way in which the variables change on each pass or ***iteration*** through the loop. Table 4-4 helps in this endeavor. On the 100th iteration, cntr is increased to 101, so there is never a 101st iteration, and we are confident that the sum is computed correctly.

ITERATION NUMBER	VALUE OF CNTR AT POINT P	VALUE OF SUM AT POINT P	VALUE OF CNTR AT POINT Q
1	1	1	2
2	2	1 + 2	3
...	...	...	...
100	100	1 + 2 +...+ 100	101

The while Loop Accumulator

Write code that computes the sum of the numbers between 1 and 10.

```
int counter = 1;
int sum = 0;
while (counter <= 10)
{
    sum = sum + counter;
    counter = counter + 1;
}
```

4.6 The while Statement

Counting Backwards

- The counter can run backward.
- The next example displays the square roots of the numbers 25, 20, 15, and 10
- Here the counter variable is called number:

```
// display the square roots of 25, 20, 15, and 10
int number = 25;
while (number >= 10)
{
System.out.println ("The square root of" + number + "is" + Math.sqrt (number));
number -= 5;
}
```

4.6 The while Statement

The output is:

```
The square root of 25 is 5.0
The square root of 20 is 4.47213595499958
The square root of 15 is 3.872983346207417
The square root of 10 is 3.162277660168379
```

4.6 The while Statement

Task-Controlled Loop

- Task-controlled loops are structured so that they continue to execute until some task is accomplished
- The following code finds the first integer for which the sum 1+2+…+n is over a million:

```
// Display the first value n for which 1+2+…+n
// Is greater than a million
int sum = 0;
int number = 0;
while (sum<= 1000000)
 {
      number++;
      sum += number;
}
system.out.println  (number);
```

4.6 The while Statement

Common Structure

- Loops typically adhere to the following structure:

```
initialize variables                    //initialize (a "primed" while loop)
while (condition)
{                                                    //test
      perform calculations and                       //loop
      change variables involved in the condition     //body
}
```

- In order for the loop to terminate, each iteration through the loop must move variables involved in the condition significantly closer to satisfying the condition.

Using while to Compute Factorials

- Example 4.2: Compute and display the factorial of *n*

```
import java.util.Scanner;

public class Factorial{

   public static void main(String[] args){
      Scanner reader = new Scanner(System.in);
      System.out.print("Enter a number greater than 0: ");
      int number = reader.nextInt();
      int product = 1;
      int count = 1;
      while (count <= number){
         product = product * count;
         System.out.println(product);
         count++;
      }
      System.out.println("The factorial of " + number +
                         " is " + product);
   }
}
```

while Loops: Discussion

- The condition can be any valid Boolean Expression
- The Boolean Expression must have a value **PRIOR** to **entering** the loop.
- The body of the loop can be a compound statement or a simple statement.
- The loop control condition needs to change in the loop body
  - If the condition is true and the condition is not changed or updated, an infinite loop could result.
  - If the condition is true and never becomes false, this results in an infinite loop also.

Error Trapping

```
//primed while loop used for error trapping a Terminal IO program.

System.out.print ("Enter a score between  ” + low_double + “ and
    “ + high_double);
score = reader.nextDouble( );
while((score < low_double) || (score > high_double))
{
     System.out.println (" Invalid score, try again.”);

    //update the value to be tested in the Boolean Expression

   System.out.print ("Enter a score between  ” + low_double + “
    and  “ + high_double);
   score = reader.nextDouble( );
}
```

4.7 The for Statement

- The for statement combines counter initialization, condition test, and update into a single expression.
- The form for the statement:

```
for  (initialize counter; test counter; update counter)
     statement;          // one statement inside the loop body

for (initialize counter; test counter; update counter)
{
     statement;          // many statements
     statement;          //inside the
     . . .;              //loop body
}
```

4.7 The for Statement

- When the for statement is executed, the counter is initialized.
- As long as the test yields true, the statements in the loop body are executed, and the counter is updated.
- The counter is updated at the bottom of the loop, after the statements in the body have been executed.

4.7 The for Statement

- The following are examples of for statements used in count-controlled loops:

```
// Compute 1 + 2 + … + 100

int sum = 0, counter;
for (counter = 1; counter <=100; counter ++)
    sum += counter;
System.out.println (sum);

// Display the square roots of 25, 20, 15, and 10
int number;
for (number = 25; number>=10; number-= 5)
    System.out.println ("The square root of"  + number +
                "is" + Math.sqrt (number));
```

4.7 The for Statement

Declaring the Loop Control Variable in a for Loop.

- The for loop allows the programmer to declare the loop control variable inside of the loop header.
- The following are equivalent loops that show these two alternatives:

```
int counter;              //Declare control variable above loop
for (counter = 1; counter <= 10; counter ++)
    System.out.println(counter);

for (int counter = 1; counter <= 10; counter ++)
     //Declare control variable in loop header
    System.out.println(counter);
```

4.7 The for Statement

- Both loops are equivalent in function, however the second alternative is considered preferable on most occasions for two reasons:

1. The loop control variable is visible only within the body of the loop where it is intended to be used.
2. The same name can be declared again in other for loops in the same program.

for loop example

- A for loop to calculate the sum of integers between a starting and ending value.

```
// Display the sum of the integers between a startingValue
// and an endingValue, using a designated increment.

int cntr, sum, startingValue, endingValue, increment;

startingValue = 10;
endingValue = 100;
increment = 7;

sum = 0;
for (cntr = startingValue; cntr <= endingValue; cntr += increment)
   sum += cntr;
System.out.println (sum);
```

User Supplied Upper Limit

//Example Program **ForUpperLimit.java** **ForUpperLimit.txt**

```
Scanner reader = new Scanner(System.in);
double number, sum = 0;
int upperLimit = 0;

System.out.print("How long is the list: ");
upperLimit = reader.nextInt();

for (int counter = 1; counter <= upperLimit; counter++)
{
   System.out.print("Enter a positive number: ");
   number = reader.nextDouble();
   sum += number;
}

System.out.println("The sum is: " + sum );
```

for vs. while Loops

- Both `for` loops and `while` loops are **entry-controlled** loops.
  - Condition tested at top of loop on each pass
- Choosing a `for` loop versus a `while` loop is often a matter of style.
  - `for` loop advantages:
    - Can declare loop control variable in header
    - Initialization, condition, and update in one line of code

4.8 Nested Control Statements and the break Statement

- Control statements can be nested inside each other in any combination that proves useful.
- The break statement can be used for breaking out of a loop early, that is before the loop condition is false.
- Break statements can be used similarly with both, for loops, and while loops.
- ***Clausen says, "Don't use a break statement in AP Computer Science!"***

4.8 Nested Control Statements and the break Statement

Print the Divisors

- As a first example, we write a code segment that asks the user for a positive integer n, and then prints all it's proper divisors, that is, all divisors except one and the number itself.
- For instance, the proper divisors of 12 are 2, 3, 4 ,and 6.
- A positive integer d is a divisor of n if d is less than n and n % d is zero. Thus, to find n's proper divisors, we must try all values of d between 2 and n / 2.

4.8 Nested Control Statements and the break Statement

Here is the code:

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

4.8 Nested Control Statements and the break Statement

Is a Number Prime?

- A number is prime if it has no proper divisors. We can modify the previous code segment to determine if a number is prime by counting it's proper divisors. If there are none, the number is prime.
- Here is the code:

4.8 Nested Control Statements and the break Statement

```
// determine if a number is prime
System.out.print ("Enter an integer greater than 2: ");
int n = reader.nextInt();
int count = 0;
int limit = n / 2;

for (int d = 2; d <= limit; d++)
{
      if (n % d == 0) //Control statements can be nested
            count++;
}
if (count != 0)
      System.out.println ("Not prime.");
else
      System.out.println ("Prime.");
```

4.8 Nested Control Statements and the break Statement

The break Statement **(Don't Use in APCS!)**

- To exit out of a loop before the loop condition is false, a break statement could be used.
- A loop either for or while, terminates immediately when a break statement is executed.
- In the following segment of code, we check d after the for loop terminates. If n has a divisor, the break statement executes, the loop terminates early, and d is less than or equal to the limit.
- **We will rewrite the sample code for "prime number test" using a while loop, instead of a for loop using break.**

Rationale for Avoiding break

- It is your instructor's belief that you should **not** use a break statement to exit out of any type of loop in AP Computer Science A.
- Any loop that exits using an if statement paired with a break statement can be rewritten using a while or do…while loop using a compound Boolean expression.
- I require that you practice using compound Boolean expressions in order to master the logic of compound expressions for the entire year.
- break statements are presented here in order that you will be able to understand programs that use this structure.
- Let's look at the source code from the textbook compared with the rewritten code:

4.8 Nested Control Statements and the break Statement

```
//Determine if a number is prime from the textbook
//This version uses a break statement (Don't use break!)
System.out.print ("Enter an integer greater than 2: ");
int n = reader.nextInt();
int limit = (int)Math.sqrt (n);
int d;                              //Declare control variable here
for (d = 2; d <= limit; d++)
{
      if (n % d == 0)
            break;
}
if (d <= limit)                     //So it's visible here
      System.out.println ("Not prime.");
else
      System.out.println ("Prime.");
```

Determine If Number is Prime (without use of break Page 125)

```
//PrimeCheckWithoutBreak.java  PrimeCheckWithoutBreak.txt
//This method is required by your teacher (Mr. Clausen)
    System.out.print ("Enter an integer greater than 2: ");
     userNumber = reader.nextInt();

     limit = (int)Math.sqrt(userNumber);

     while((divisor<=limit) && (userNumber % divisor != 0))
     {
       divisor++;
     }
     // Display whether userNumber is prime or not prime
     if (divisor <= limit)
        System.out.println ("Not prime");
     else
        System.out.println ("Prime"); //This code uses the same number of lines.
```

Sentinel-controlled input

- **Sentinel-controlled input:** Continue a loop until a **sentinel** variable has a specific value

```
Scanner reader = new Scanner(System.in);
double number, sum = 0;
int count = 0;

while (true){
   System.out.print("Enter a positive number or -1 to quit: ");
   number = reader.nextDouble();
   if (number == -1) break;
   sum += number;
   count++;
}

if (count == 0)
   System.out.println ("The list is empty.");
else
   System.out.println ("The average is " + sum / count);
```

Sentinel Values and Counters

- Sentinel Value

DemoSentinel.java DemoSentinel.txt

The code listed above is rewritten to avoid using a break statement. Use this instead of the textbook's approach.

- A value that determines the end of a set of data, or the end of a process in an indefinite loop.
- While loops may be repeated an indefinite number of times.
  - It is common to count the number of times the loop repeats.
  - Initialize this "counter" before the loop
  - Increment the counter inside the loop

Using Loops with Text Files

- Advantages of using text files versus input from a human user:
  - Data sets can be much larger.
  - Data input quickly, with less chance of error.
  - Data can be used repeatedly.
- Data files can be created by hand in a text editor or generated by a program.

Using Loops with Text Files (cont.) 2

- Text file format:
  - If data is numerical, numbers must be separated by white space.
  - Must be an **end-of-file** character
    - Used by program to determine whether it is done reading data
- When an error occurs at run-time, the JVM **throws** an **exception**.
  - `IOException` thrown if error occurs during file operations
  - **import java.io.*;** //For file and IOException

Using Loops with Text Files (cont.) 3

Example 4.3: Computes and displays the average of a file of floating-point numbers

```
import java.io.*;                    // For File and IOException
import java.util.Scanner;
```

Using Loops with Text Files (cont.) 4

Example 4.4: Inputs a text file of integers and writes these to an output file without the zeroes

```
import java.io.*;           // For File, IOException, and PrintWriter
import java.util.Scanner;

public class FilterZeroes{

   public static void main(String[] args) throws IOException {

      // Open the scanner and print writer
      Scanner reader = new Scanner(new File("numbers.txt"));
      PrintWriter writer = new PrintWriter(new File("newnumbers.txt"));

      // Read the numbers and write all but the zeroes
      while (reader.hasNext()){
         int number = reader.nextInt();
         if (number != 0)
            writer.println(number);
      }

      // Remember to close the output file
      writer.close();
   }
}
```

Text File Program Examples

- Example 4.3 Input from a text file
  - ComputeAverage.java
  - ComputeAverage.txt
- Example 4.4 Output to a text file
  - FilterZeros.java
  - FilterZeros.txt
- The data file: numbers.txt

Case Study:
The Folly of Gambling

LuckySevens.java

LuckySevens.txt

4.9 Errors in Loops

A loop usually has four component parts:

- **Initializing Statements.**
  - These statements initialize variables used within the loop.
- **Terminating condition.**
  - This condition is tested before each pass through the loop to determine if another iteration is needed.
- **Body Statements.**
  - these statements execute on each iteration and implement the calculation in question.

4.9 Errors in Loops

- **Update statements.**
  - These statements, which usually are executed at the bottom of the loop, change the values of the variables tested in the terminating condition.

- A careless programmer can introduce logic errors into any of these components.

4.9 Errors in Loops

Initialization Error

- Because we forget to initialize the variable product, it retains its default value of zero.

```
//Error – failure to initialize the variable product
//Outcome – zero is printed
int product;       //correct:  int product = 1;
i = 3;
while (i <= 100)
{
      product = product * i;
      i = i + 2;
 }
System.out.println (product);
```

4.9 Errors in Loops

Off-by-One Error

- The ***off-by-one error***, occurs whenever a loop goes around one too many or one too few times
- This is one of the most common types of looping errors and is often difficult to detect.

4.9 Errors in Loops

```
//Error – use of  "< 99" rather than "<= 100" in the
//              terminating condition
//Outcome – product will equal  3 * 5...* 97
product = 1;
i = 3;
while  (i < 99)  //Correct ( i <= 100)
{
product = product * i;
i = i +2;
}
System.out.println (product);
```

4.9 Errors in Loops

Infinite Loop

```
//Error – use of  "!= 100" rather than  "<= 100" in
//the terminating condition.
// Outcome – the program will never stop
product = 1;
i = 3;
while (i   != 100)   //Correct (i   <= 100)
{
     product = product * i;
     i = i + 2;
}
System.out.println (product);
```

4.9 Errors in Loops

- The variable i takes on the values 3, 5, …., 99, 101, … and never equals 100.
- This is called an ***infinite loop***.
- Anytime a program responds more slowly than expected, it is reasonable to assume that it is stuck in an infinite loop.
- To stop the program select the terminal window and type Ctrl+C.

4.9 Errors in Loops

Update Error

- If the update statement is in the wrong place, the calculations can be thrown off even if the loop iterates the correct number of times:

```
//Error – placement of the update statement in the wrong place
//Outcome – product will equal 5*7*...*99*101
product = 1;
i = 3;
while (i <= 100){
   i = i + 2;  //this update statement should follow the calculation of product
   product = product * i;
}
System.out.println (product);
```

4.9 Errors in Loops

- **Effects of limited floating-point precision:** When using floating-point variables as loop control variables, you must understand that not all values can be represented.
  - Better **not** to use == or != in condition statement under these conditions

4.9 Errors in Loops

Debugging Loops

- If you suspect that you have written a loop that contains a logic error, inspect the code and make sure the following items are true:
  - Variables are initialized correctly before entering the loop.
  - the terminating condition stops the iterations when the test variables have reached the intended limit.

4.9 Errors in Loops

- The statements in the body are correct.
- The update statements are positioned correctly and modify the test variables in such a manner that they eventually pass the limits tested in the terminating condition.
- When writing terminating conditions, it is usually safer to use one of the operators:

  `<` `<=` `>` `>=`

  than either of the operators

  `==` `!=`

4.9 Errors in Loops

- If you cannot find an error by inspection, then use System.out.println statements to "dump" key variables to the terminal window.
- Good places for these statements are:
  - Immediately after the initialization statements.
  - Inside the loop at the top.
  - Inside the loop at the bottom.
- You will then discover that some of the variables have values different than expected, and this will provide clues that reveal the exact nature and location of the logic error.

Debugging Research

"If the source of the problem is not immediately obvious, leave the computer and go somewhere where you can quietly look over a printed copy of the program. Studies show that people who do all of their debugging away from the computer actually get their programs to work in less time and in the end produce better programs than those who continue to work on the machine-more proof that there is still no mechanical substitute for human thought."*

Dale, Weams, Headington "Programming and Problem Solving with C++", Jones and Bartlett Publishers, 1997, pp80-81

Basili, V.R., Selby, R.W., "Comparing the Effectiveness of Software Testing Strategies",IEEE Trans. On Software Engineering, Vol. SE-13, No.12, pp 1278-1296, Dec. 1987

Design, Testing, and Debugging Hints

- Most errors involving selection statements and loops are not syntax errors.
- The presence or absence of the `{}` symbols can seriously affect the logic of a selection statement or loop.
- When testing programs that use `if` or `if-else` statements, use test data that forces the program to exercise all logical branches.

Design, Testing, and Debugging Hints (cont.)

- Use an `if-else` statement rather than two `if` statements when the alternative courses of action are mutually exclusive.
- When testing a loop, be sure to use limit values as well as typical values.
- Be sure to check entry conditions and exit conditions for each loop.

Design, Testing, and Debugging Hints (cont.)

- For a loop with errors, use debugging output statements to verify the values of the control variable on each pass through the loop.
- Text files are convenient to use when the data set is large, when the same data set must be used repeatedly with different programs, and when these data must be saved permanently.

Summary

- Java has operators for extended assignment and for increment and decrement.
- The `Math` class provides several useful methods, such as `sqrt` and `abs`.
- The `Random` class allows you to generate random integers and floating-point numbers.
- `if` and `if-else` statements are used to make one-way and two-way decisions.

Summary (cont.)

- The comparison operators, such as ==, <=, and >=, return Boolean values that can serve as conditions for control statements.
- The `while` loop allows the program to run a set of statements repeatedly until a condition becomes false.
- The `for` loop is a more concise version of the `while` loop.

Summary (cont.)

- Other control statements, such as an `if` statement, can be nested within loops.
- A `break` statement can be used with an `if` statement to terminate a loop early.
- Many kinds of logic errors can occur in loops.
  - Off-by-one error
  - Infinite loop