Java Program Statements

Selim Aksoy
Bilkent University
Department of Computer Engineering
saksoy@cs.bilkent.edu.tr

Program Development

- The creation of software involves four basic activities:
  - establishing the requirements
  - creating a design
  - implementing the code
  - testing the implementation
- The development process is much more involved than this, but these are the four basic development activities

Requirements

- *Software requirements* specify the tasks a program must accomplish (what to do, not how to do it)
- They often include a description of the user interface
- An initial set of requirements often are provided, but usually must be critiqued, modified, and expanded
- Often it is difficult to establish detailed, unambiguous, complete requirements
- Careful attention to the requirements can save significant time and expense in the overall project

Design

- A *software design* specifies how a program will accomplish its requirements
- A design includes one or more *algorithms* to accomplish its goal
- An *algorithm* is a step-by-step process for solving a problem
- An algorithm may be expressed in *pseudocode*, which is code-like, but does not necessarily follow any specific syntax
- In object-oriented development, the design establishes the classes, objects, methods, and data that are required

Implementation

- *Implementation* is the process of translating a design into source code
- Most novice programmers think that writing code is the heart of software development, but actually it should be the least creative step
- Almost all important decisions are made during requirements and design stages
- Implementation should focus on coding details, including style guidelines and documentation

Testing

- A program should be executed multiple times with various input in an attempt to find errors
- *Debugging* is the process of discovering the causes of problems and fixing them
- Programmers often think erroneously that there is "only one more bug" to fix
- Tests should consider design details as well as overall requirements

Flow of Control

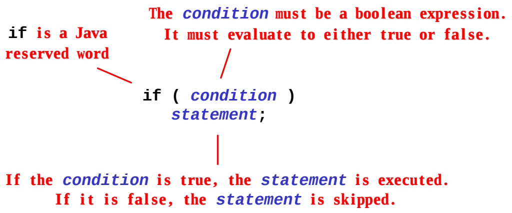
- Unless specified otherwise, the order of statement execution through a method is linear: one statement after the other in sequence
- Some programming statements modify that order, allowing us to:
  - decide whether or not to execute a particular statement, or
  - perform a statement over and over, repetitively
- These decisions are based on a *boolean expression* (also called a *condition*) that evaluates to true or false
- The order of statement execution is called the *flow of control*

Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- Therefore they are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- Java's conditional statements are
  - the *if statement*
  - the *if-else statement*
  - the *switch statement*

The if Statement

- The *if statement* has the following syntax:



if is a Java reserved word

The *condition* must be a boolean expression. It must evaluate to either true or false.

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

If the *condition* is true, the *statement* is executed. If it is false, the *statement* is skipped.

The if Statement

- An example of an `if` statement:

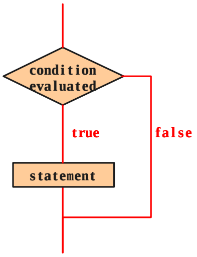
```
if (sum > MAX)
   delta = sum - MAX;
System.out.println ("The sum is " + sum);
```

First, the condition is evaluated. The value of sum is either greater than the value of MAX, or it is not.

If the condition is true, the assignment statement is execu
If it is not, the assignment statement is skipped.

Either way, the call to println is executed next.

Logic of an if statement



Boolean Expressions

- A condition often uses one of Java's *equality operators* or *relational operators*, which all return boolean results:

==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

- Note the difference between the equality operator (==) and the assignment operator (=)

The if-else Statement

- An *else clause* can be added to an `if` statement to make an *if-else statement*

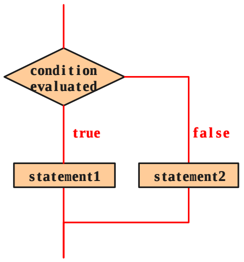
```
        if ( condition )
            statement1;
        else
            statement2;
```

- If the `condition` is true, `statement1` is executed; if the condition is false, `statement2` is executed
- One or the other will be executed, but not both

Example

```
import java.text.NumberFormat ;
import cs1.Keyboard;

public class Wages
{
   //-----------------------------------------------------------------
   //  Reads the number of hours worked and calculates wages.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final double RATE = 8.25;  // regular pay rate
      final int STANDARD = 40;   // standard hours in a work week

      double pay = 0.0;

      System.out.print ("Enter the number of hours worked: ");
      int hours = Keyboard.readInt ();

      System.out.println ();

      // Pay overtime at "time and a half"
      if (hours > STANDARD)
         pay = STANDARD * RATE + (hours -STANDARD) * (RATE * 1.5);
      else
         pay = hours * RATE;

      NumberFormat  fmt = NumberFormat.getCurrencyInstance( );
      System.out.println ("Gross earnings: " + fmt.format(pay ));
   }
}
```

Logic of an if-else statement



Block Statements

- Several statements can be grouped together into a *block statement*
- A block is delimited by braces : { … }
- A block statement can be used wherever a statement is called for by the Java syntax
- For example, in an `if-else` statement, the `if` portion, or the `else` portion, or both, could be block statements

Example

```
import cs1.Keyboard;
import java.util.Random ;

public class Guessing
{
   //-----------------------------------------------------------------
   //  Plays a simple guessing game with the user.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int MAX = 10;
      int answer, guess;

      Random generator = new Random();
      answer = generator.nextInt(MAX) + 1;

      System.out.print ("I'm thinking of a number between 1 and "
                        + MAX + ". Guess what it is: ");
      guess = Keyboard.readInt();

      if (guess == answer)
         System.out.println ("You got it! Good guessing!");
      else
      {
         System.out.println ("That is not correct, sorry.");
         System.out.println ("The number was " + answer);
      }
   }
}
```

Nested if Statements

- The statement executed as a result of an `if` statement or `else` clause could be another `if` statement
- These are called *nested if statements*
- An `else` clause is matched to the last unmatched `if` (no matter what the indentation implies)
- Braces can be used to specify the `if` statement to which an `else` clause belongs

Example

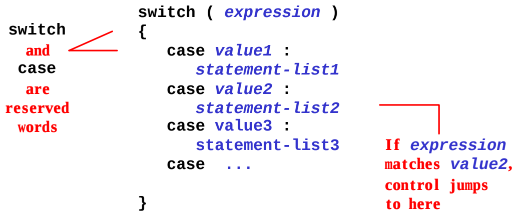
```
import cs1.Keyboard;

public class MinOfThree
{
   //-----------------------------------------------------------------
   //  Reads three integers from the user and determines the smallest
   //  value.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int num1, num2, num3, min = 0;

      System.out.println ("Enter three integers: ");
      num1 = Keyboard.readInt();
      num2 = Keyboard.readInt();
      num3 = Keyboard.readInt();

      if (num1 < num2)
         if (num1 < num3)
            min = num1;
         else
            min = num3;
      else
         if (num2 < num3)
            min = num2;
         else
            min = num3;

      System.out.println ("Minimum value: " + min);
   }
}
```

The switch Statement

- The *switch statement* provides another means to decide which statement to execute next
- The `switch` statement evaluates an expression, then attempts to match the result to one of several possible *cases*
- Each case contains a value and a list of statements
- The flow of control transfers to statement associated with the first value that matches

The switch Statement

- The general syntax of a `switch` statement is:



```
switch ( expression )
{
   case value1 :
      statement-list1
   case value2 :
      statement-list2
   case value3 :
      statement-list3
   case   ...

}
```

switch and case are reserved words

If expression matches value2, control jumps to here

The switch Statement

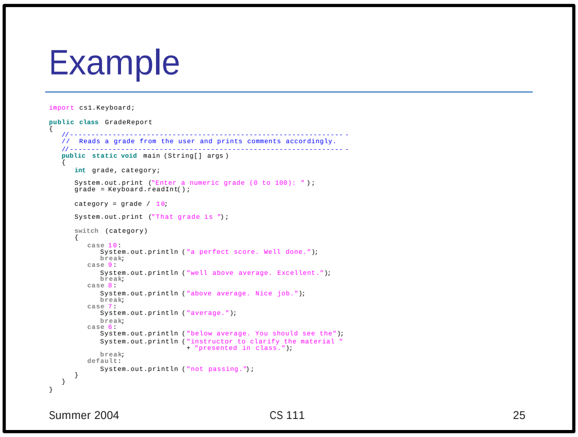
- Often a *break statement* is used as the last statement in each case's statement list
- A `break` statement causes control to transfer to the end of the `switch` statement
- If a `break` statement is not used, the flow of control will continue into the next case
- Sometimes this can be appropriate, but usually we want to execute only the statements associated with one case

The switch Statement

- A `switch` statement can have an optional *default case*
- The default case has no associated value and simply uses the reserved word `default`
- If the default case is present, control will transfer to it if no other case value matches
- Though the default case can be positioned anywhere in the switch, usually it is placed at the end
- If there is no default case, and no other value matches, control falls through to the statement after the switch

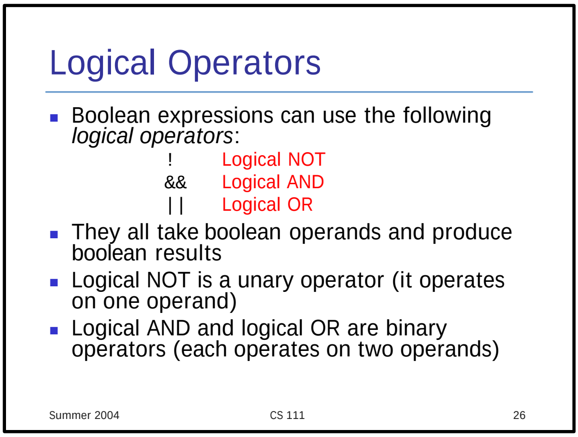
The switch Statement

- The expression of a `switch` statement must result in an *integral type*, meaning an `int` or a `char`
- It cannot be a `boolean` value, a floating point value (`float` or `double`), a `byte`, a `short`, or a `long`
- The implicit boolean condition in a `switch` statement is equality - it tries to match the expression with a value
- You cannot perform relational checks with a `switch` statement



Example

```
import cs1.Keyboard;

public class GradeReport
{
   //-----------------------------------------------------------------
   //  Reads a grade from the user and prints comments accordingly.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int grade, category;

      System.out.print ("Enter a numeric grade (0 to 100): ");
      grade = Keyboard.readInt();

      category = grade / 10;

      System.out.print ("That grade is ");

      switch (category)
      {
         case 10:
            System.out.println ("a perfect score. Well done.");
            break;
         case 9:
            System.out.println ("well above average. Excellent.");
            break;
         case 8:
            System.out.println ("above average. Nice job.");
            break;
         case 7:
            System.out.println ("average.");
            break;
         case 6:
            System.out.println ("below average. You should see the");
            System.out.println ("instructor to clarify the material "
                                + "presented in class.");
            break;
         default:
            System.out.println ("not passing.");
      }
   }
}
```

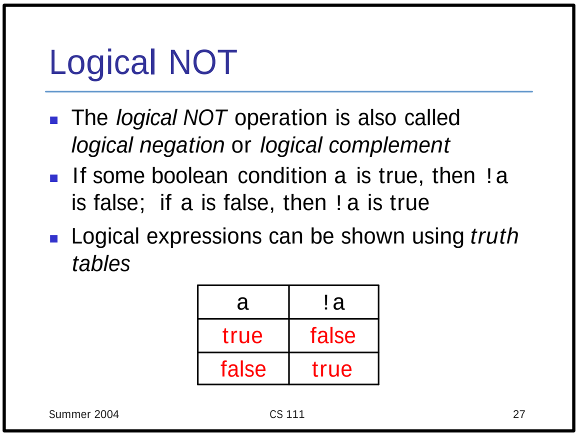


Logical Operators

- Boolean expressions can use the following *logical operators*:

!	Logical NOT
&&	Logical AND
| |	Logical OR

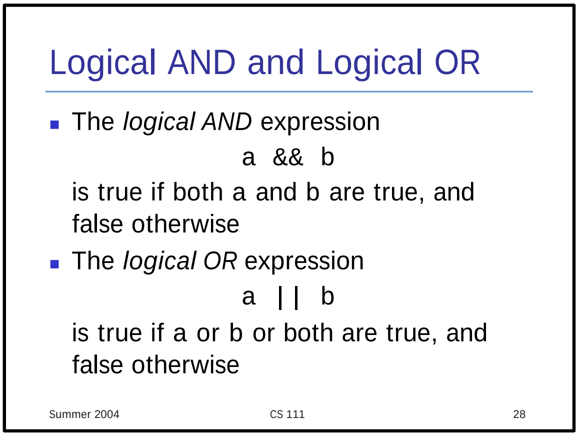
- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)



Logical NOT

- The *logical NOT* operation is also called *logical negation* or *logical complement*
- If some boolean condition a is true, then !a is false; if a is false, then !a is true
- Logical expressions can be shown using *truth tables*

a	!a
true	false
false	true



Logical AND and Logical OR

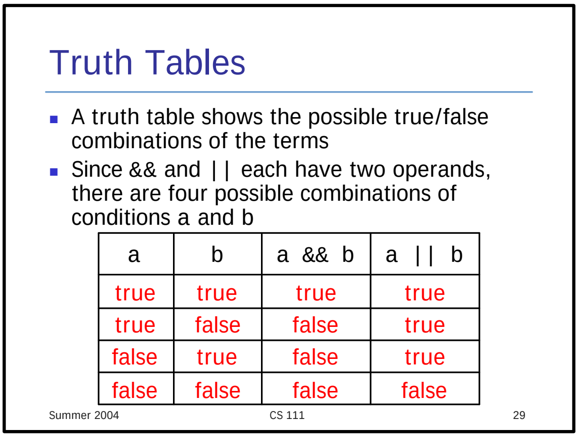
- The *logical AND* expression

  a && b

  is true if both a and b are true, and false otherwise
- The *logical OR* expression

  a || b

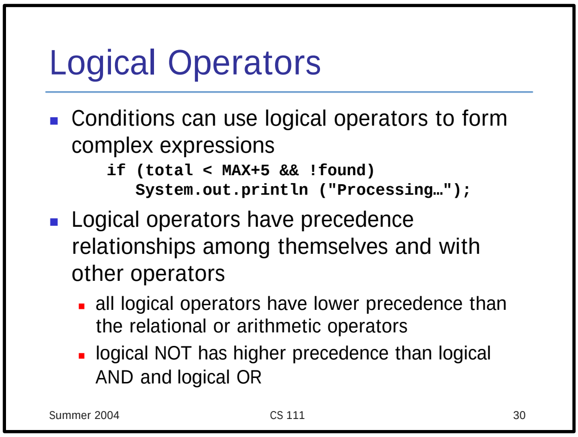
  is true if a or b or both are true, and false otherwise



Truth Tables

- A truth table shows the possible true/false combinations of the terms
- Since && and || each have two operands, there are four possible combinations of conditions a and b

a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false



Logical Operators

- Conditions can use logical operators to form complex expressions

```
if (total < MAX+5 && !found)
   System.out.println ("Processing…");
```

- Logical operators have precedence relationships among themselves and with other operators
  - all logical operators have lower precedence than the relational or arithmetic operators
  - logical NOT has higher precedence than logical AND and logical OR

Short Circuited Operators

- The processing of logical AND and logical OR is “short-circuited”
- If the left operand is sufficient to determine the result, the right operand is not evaluated

```
    if (count != 0 && total/count > MAX)
       System.out.println ("Testing…");
```

- This type of processing must be used carefully

Truth Tables

- Specific expressions can be evaluated using truth tables

total < MAX	found	!found	total < MAX && !found
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

Comparing Characters

- We can use the relational operators on character data
- The results are based on the Unicode character set
- The following condition is true because the character + comes before the character J in the Unicode character set:

```
    if ('+' < 'J')
       System.out.println ("+ is less than J");
```

- The uppercase alphabet (A-Z) followed by the lowercase alphabet (a-z) appear in alphabetical order in the Unicode character set

Comparing Strings

- Remember that a character string in Java is an object
- We cannot use the relational operators to compare strings
- The `equals` method can be called with strings to determine if two strings contain exactly the same characters in the same order
- The `String` class also contains a method called `compareTo` to determine if one string comes before another (based on the Unicode character set)

Lexicographic Ordering

- Because comparing characters and strings is based on a character set, it is called a *lexicographic ordering*
- This is not strictly alphabetical when uppercase and lowercase characters are mixed
- For example, the string `"Great"` comes before the string `"fantastic"` because all of the uppercase letters come before all of the lowercase letters in Unicode
- Also, short strings come before longer strings with the same prefix (lexicographically)
- Therefore `"book"` comes before `"bookcase"`

Comparing Float Values

- We also have to be careful when comparing two floating point values (`float` or `double`) for equality
- You should rarely use the equality operator (==) when comparing two floats
- In many situations, you might consider two floating point numbers to be "close enough" even if they aren't exactly equal
- Therefore, to determine the equality of two floats, you may want to use the following technique:

```
    if (Math.abs(f1 - f2) < 0.00001)
       System.out.println ("Essentially equal.");
```

More Operators

- To round out our knowledge of Java operators, let's examine a few more
- In particular, we will examine
  - the increment and decrement operators
  - the assignment operators
  - the conditional operator

Increment and Decrement

- The increment and decrement operators are arithmetic and operate on one operand
- The *increment operator* (++) adds one to its operand
- The *decrement operator* (--) subtracts one from its operand
- The statement

  ```
  count++;
  ```

  is functionally equivalent to

  ```
  count = count + 1;
  ```

Increment and Decrement

- The increment and decrement operators can be applied in *prefix form* (before the operand) or *postfix form* (after the operand)
- When used alone in a statement, the prefix and postfix forms are functionally equivalent. That is,

  ```
  count++;
  ```

  is equivalent to

  ```
  ++count;
  ```

Increment and Decrement

- When used in a larger expression, the prefix and postfix forms have different effects
- In both cases the variable is incremented (decremented)
- But the value used in the larger expression depends on the form used:

Expression	Operation	Value Used in Expression
count++	**add 1**	**old value**
++count	**add 1**	**new value**
count--	**subtract 1**	**old value**
--count	**subtract 1**	**new value**

Increment and Decrement

- If `count` currently contains 45, then the statement

  ```
  total = count++;
  ```

  assigns 45 to `total` and 46 to `count`
- If `count` currently contains 45, then the statement

  ```
  total = ++count;
  ```

  assigns the value 46 to both `total` and `count`

Assignment Operators

- Often we perform an operation on a variable, and then store the result back into that variable
- Java provides *assignment operators* to simplify that process
- For example, the statement

  ```
  num += count;
  ```

  is equivalent to

  ```
  num = num + count;
  ```

Assignment Operators

- There are many assignment operators, including the following:

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Assignment Operators

- The right hand side of an assignment operator can be a complex expression
- The entire right-hand expression is evaluated first, then the result is combined with the original variable
- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```

Assignment Operators

- The behavior of some assignment operators depends on the types of the operands
- If the operands to the += operator are strings, the assignment operator performs string concatenation
- The behavior of an assignment operator (+=) is always consistent with the behavior of the "regular" operator (+)

The Conditional Operator

- Java has a *conditional operator* that evaluates a boolean condition that determines which of two other expressions is evaluated
- The result of the chosen expression is the result of the entire conditional operator
- Its syntax is:

  `condition` `?` *`expression1`* `:` *`expression2`*
- If the *`condition`* is true, *`expression1`* is evaluated; if it is false, *`expression2`* is evaluated

The Conditional Operator

- The conditional operator is similar to an `if-else` statement, except that it forms an expression that returns a value
- For example:

```
larger = ((num1 > num2) ? num1 : num2);
```

- If `num1` is greater that `num2`, then `num1` is assigned to `larger`; otherwise, `num2` is assigned to `larger`
- The conditional operator is *ternary* because it requires three operands

The Conditional Operator

- Another example:

```
System.out.println("Your change is " + count +
    ((count == 1) ? "Dime" : "Dimes"));
```

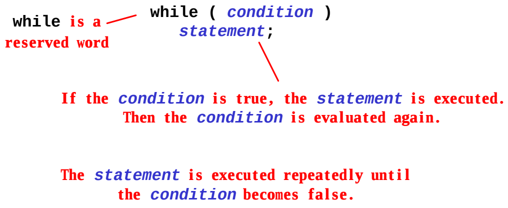
- If `count` equals 1, then `"Dime"` is printed
- If `count` is anything other than 1, then `"Dimes"` is printed

Repetition Statements

- *Repetition statements* allow us to execute a statement multiple times
- Often they are referred to as *loops*
- Like conditional statements, they are controlled by boolean expressions
- Java has three kinds of repetition statements:
  - the *while loop*
  - the *do loop*
  - the *for loop*
- The programmer should choose the right kind of loop for the situation

The while Statement

- The *while statement* has the following syntax:



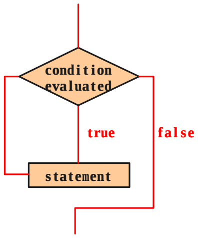
```
while ( condition )
   statement;
```

while is a reserved word

If the condition is true, the statement is executed. Then the condition is evaluated again.

The statement is executed repeatedly until the condition becomes false.

Logic of a while Loop



The while Statement

- Note that if the condition of a `while` statement is false initially, the statement is never executed
- Therefore, the body of a `while` loop will execute zero or more times

Example

```
//********************************************************************
//  Counter.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a while loop.
//********************************************************************

public class Counter
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;
      int count = 1;

      while (count <= LIMIT)
      {
         System.out.println (count);
         count = count + 1;
      }

      System.out.println ("Done");
   }
}
```

Example

A *sentinel value* indicates the end of the input
The variable sum maintains a *running sum*

```
import java.text.DecimalFormat;
import cs1.Keyboard;

public class Average
{
   //-----------------------------------------------------------------
   //  Computes the average of a set of values entered by the user.
   //  The running sum is printed as the numbers are entered.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int sum = 0, value, count = 0;
      double average;

      System.out.print ("Enter an integer (0 to quit): ");
      value = Keyboard.readInt();

      while (value != 0)  // sentinel value of 0 to terminate loop
      {
         count++;

         sum += value;
         System.out.println ("The sum so far is " + sum);

         System.out.print ("Enter an integer (0 to quit): ");
         value = Keyboard.readInt();
      }

      System.out.println ();
      System.out.println ("Number of values entered: " + count);

      average = (double)sum / count;

      DecimalFormat fmt = new DecimalFormat ("0.###");
      System.out.println ("The average is " + fmt.format(average));
   }
}
```

Example

A loop is used to *validate the input*, making the program more *robust*

```
import java.text.NumberFormat;
import cs1.Keyboard;

public class WinPercentage
{
   //-----------------------------------------------------------------
   //  Computes the percentage of games won by a team.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int NUM_GAMES = 12;
      int won;
      double ratio;

      System.out.print ("Enter the number of games won (0 to "
                        + NUM_GAMES + "): ");
      won = Keyboard.readInt();

      while (won < 0 || won > NUM_GAMES)
      {
         System.out.print ("Invalid input. Please reenter: ");
         won = Keyboard.readInt();
      }

      ratio = (double)won / NUM_GAMES;

      NumberFormat fmt = NumberFormat.getPercentInstance();

      System.out.println ();
      System.out.println ("Winning percentage: " + fmt.format(ratio));
   }
}
```

Infinite Loops

- The body of a `while` loop eventually must make the condition false
- If not, it is an *infinite loop*, which will execute until the user interrupts the program
- This is a common logical error
- You should always double check to ensure that your loops will terminate normally

Example

```
//********************************************************************
//  Forever.java       Author: Lewis/Loftus
//
//  Demonstrates an INFINITE LOOP.  WARNING!!
//********************************************************************

public class Forever
{
   //-----------------------------------------------------------------
   //  Prints ever decreasing integers in an INFINITE LOOP!
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int count = 1;

      while (count <= 25)
      {
         System.out.println (count);
         count = count - 1;
      }

      System.out.println ("Done");  // this statement is never reached
   }
}
```

Nested Loops

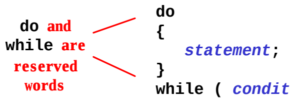
- Similar to nested `if` statements, loops can be nested as well
- That is, the body of a loop can contain another loop
- Each time through the outer loop, the inner loop goes through its full set of iterations

Example

```
import cs1.Keyboard;

public class PalindromeTester
{
   //-----------------------------------------------------------------
   //  Tests strings to see if they are palindromes.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      String str, another = "y";
      int left, right;

      while (another.equalsIgnoreCase("y")) // allows y or Y
      {
         System.out.println ("Enter a potential palindrome:");
         str = Keyboard.readString();

         left = 0;
         right = str.length() - 1;

         while (str.charAt(left) == str.charAt(right) && left < right)
         {
            left++;
            right--;
         }

         System.out.println();

         if (left < right)
            System.out.println ("That string is NOT a palindrome.");
         else
            System.out.println ("That string IS a palindrome.");

         System.out.println();
         System.out.print ("Test another palindrome (y/n)? ");
         another = Keyboard.readString();
      }
   }
}
```

The do Statement

- The *do statement* has the following syntax:



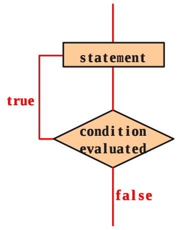
do and while are reserved words

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

The *statement* is executed once initially, and then the *condition* is evaluated

The *statement* is executed repeatedly until the *condition* becomes false

Logic of a do Loop



The do Statement

- A `do` loop is similar to a `while` loop, except that the condition is evaluated after the body of the loop is executed
- Therefore the body of a `do` loop will execute at least once

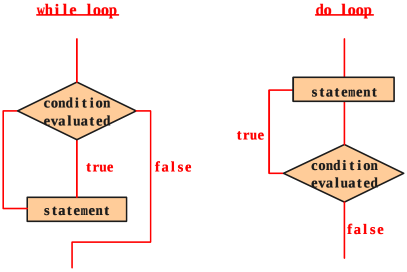
Example

```
//********************************************************************
//  Counter2.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a do loop.
//********************************************************************

public class Counter2
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;
      int count = 0;

      do
      {
         count = count + 1;
         System.out.println (count);
      }
      while (count < LIMIT);

      System.out.println ("Done");
   }
}
```

Example

```
import cs1.Keyboard;

public class ReverseNumber
{
   //-----------------------------------------------------------------
   //  Reverses the digits of an integer mathematically.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      int number, lastDigit, reverse = 0;

      System.out.print ("Enter a positive integer: ");
      number = Keyboard.readInt();

      do
      {
         lastDigit = number % 10;
         reverse = (reverse * 10) + lastDigit;
         number = number / 10;
      }
      while (number > 0);

      System.out.println ("That number reversed is " + reverse);
   }
}
```

Comparing while and do



The for Statement

- The *for statement* has the following syntax:

Reserved word

The *initialization* is executed once before the loop begins

The *statement* is executed until the *condition* becomes false

for (*initialization* ; *condition* ; *increment*)
   ***statement*;**

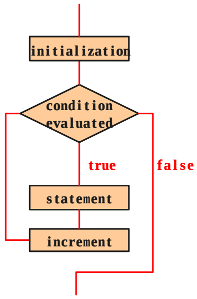
The *increment* portion is executed at the end of each iteration
The *condition-statement-increment* cycle is executed repeatedly

The for Statement

- A `for` loop is functionally equivalent to the following `while` loop structure:

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

Logic of a for loop



The for Statement

- Like a `while` loop, the condition of a `for` statement is tested prior to executing the loop body
- Therefore, the body of a `for` loop will execute zero or more times
- It is well suited for executing a loop a specific number of times that can be determined in advance

Example

```
//********************************************************************
//  Counter3.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a for loop.
//********************************************************************

public class Counter3
{
   //-----------------------------------------------------------------
   //  Prints integer values from 1 to a specific limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int LIMIT = 5;

      for (int count=1; count <= LIMIT; count++)
         System.out.println (count);

      System.out.println ("Done");
   }
}
```

Example

```
import cs1.Keyboard;

public class Multiples
{
   //-----------------------------------------------------------------
   //  Prints multiples of a user-specified number up to a user-
   //  specified limit.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int PER_LINE = 5;
      int value, limit, mult, count = 0;

      System.out.print ("Enter a positive value: ");
      value = Keyboard.readInt();

      System.out.print ("Enter an upper limit: ");
      limit = Keyboard.readInt();

      System.out.println ();
      System.out.println ("The multiples of " + value + " between " +
                          value + " and " + limit + " (inclusive) are:");

      for (mult = value; mult <= limit; mult += value)
      {
         System.out.print (mult + "\t");

         // Print a specific number of values per line of output
         count++;
         if (count % PER_LINE == 0)
            System.out.println();
      }
   }
}
```

Example

```
//********************************************************************
//  Stars.java       Author: Lewis/Loftus
//
//  Demonstrates the use of nested for loops.
//********************************************************************

public class Stars
{
   //-----------------------------------------------------------------
   //  Prints a triangle shape using asterisk (star) characters.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int MAX_ROWS = 10;

      for (int row = 1; row <= MAX_ROWS; row++)
      {
         for (int star = 1; star <= row; star++)
            System.out.print ("*");

         System.out.println();
      }
   }
}
```

The for Statement

- Each expression in the header of a for loop is optional
  - If the *`initialization`* is left out, no initialization is performed
  - If the *`condition`* is left out, it is always considered to be true, and therefore creates an infinite loop
  - If the *`increment`* is left out, no increment operation is performed
- Both semi-colons are always required in the `for` loop header

Choosing a Loop Structure

- When you can't determine how many times you want to execute the loop body, use a `while` statement or a do statement
  - If it might be zero or more times, use a `while` statement
  - If it will be at least once, use a do statement
- If you can determine how many times you want to execute the loop body, use a `for` statement

Program Development

- We now have several additional statements and operators at our disposal
- Following proper development steps is important
- Suppose you were given some initial requirements:
  - accept a series of test scores
  - compute the average test score
  - determine the highest and lowest test scores
  - display the average, highest, and lowest test scores

Program Development

- Requirements Analysis – clarify and flesh out specific requirements
  - How much data will there be?
  - How should data be accepted?
  - Is there a specific output format required?
- After conferring with the client, we determine:
  - the program must process an arbitrary number of test scores
  - the program should accept input interactively
  - the average should be presented to two decimal places
- The process of requirements analysis may take a long time

Program Development

- Design – determine a possible general solution
  - Input strategy? (Sentinel value?)
  - Calculations needed?
- An initial algorithm might be expressed in pseudocode
- Multiple versions of the solution might be needed to refine it
- Alternatives to the solution should be carefully considered

Program Development

- Implementation – translate the design into source code
- Make sure to follow coding and style guidelines
- Implementation should be integrated with compiling and testing your solution
- This process mirrors a more complex development model we'll eventually need to develop more complex software
- The result is a final implementation

Example

```
import java.text.DecimalFormat ;
import cs1.Keyboard;

public class ExamGrades
{
   public static void main (String[] args )
   {
      int grade, count = 0, sum = 0, max, min;
      double average;

      //  Get the first grade and give max and min that initial value
      System.out.print ("Enter the first grade (-1 to quit): " );
      grade = Keyboard.readInt ();

      max = min = grade;

      //  Read and process the rest of the grades
      while (grade >= 0)
      {
         count++;
         sum += grade;

         if (grade > max)
            max = grade;
         else
            if (grade < min)
               min = grade;

         System.out.print ("Enter the next grade (-1 to quit): ");
         grade = Keyboard.readInt ();
      }

      //  Produce the final results
      if (count == 0)
         System.out.println ("No valid grades were entered.");
      else
      {
         DecimalFormat fmt = new DecimalFormat ("0.##" );
         average = (double)sum / count;
         System.out.println ();
         System.out.println ("Total number of students: " + count);
         System.out.println ("Average grade: " + fmt.format(average ));
         System.out.println ("Highest grade: " + max);
         System.out.println ("Lowest grade: " + min);
      }
   }
}
```

Program Development

- Testing – attempt to find errors that may exist in your programmed solution
- Compare your code to the design and resolve any discrepancies
- Determine test cases that will stress the limits and boundaries of your solution
- Carefully retest after finding and fixing an error