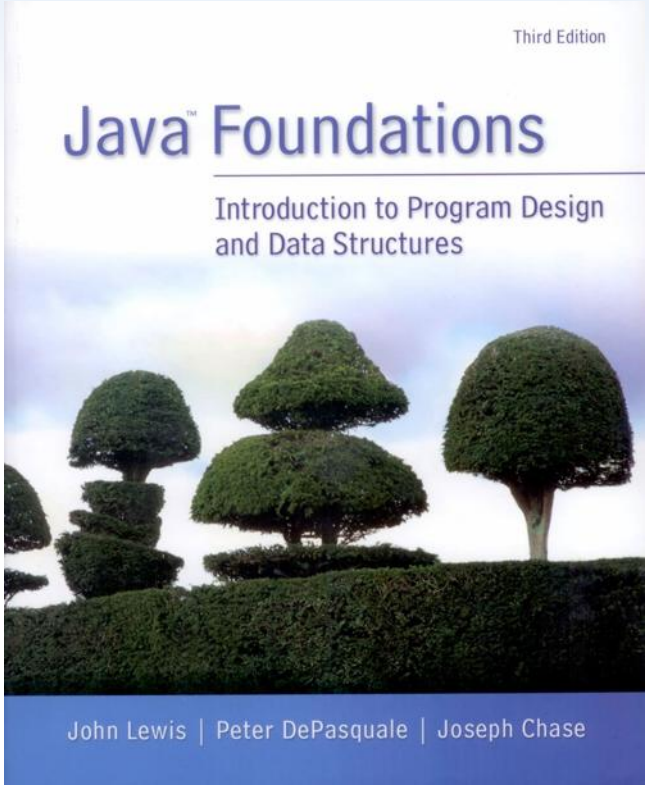



# Chapter 4

# Conditionals and Loops

# Chapter Scope

- Flow of control
- Boolean expressions
- if and switch statements
- Comparing data
- while, do, and for loops
- Iterators

# Flow of Control

- Statement execution is linear unless specified otherwise
- Some programming statements allow us to:
  - decide whether or not to execute a particular statement
  - execute a statement over and over, repetitively
- These decisions are based on *boolean expressions* (or *conditions*) that evaluate 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
- The Java conditional statements are the
  - *if statement*
  - *if-else statement*
  - *switch statement*

# The if Statement

- The syntax of a basic if statement is:

`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.

# Equality and Relational Operators

- Often, conditions are based *equality operators* or *relational operators:*

| Operator | Meaning |
|---|---|
| == | equal to |
| != | not equal to |
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |

# Conditions

- Examples of if statements:

```
if (total == sum)
    System.out.println("total equals sum");

if (count > 50)
    System.out.println("count is more than 50");

if (letter != 'x')
    System.out.println("letter is not x");
```

# Logical Operators

- Conditions can also use *logical operators*:

| Operator | Description | Example | Result |
|---|---|---|---|
| ! | logical NOT | ! a | true if a is false and false if a is true |
| && | logical AND | a && b | true if a and b are both true and false otherwise |
| \|\| | logical OR | a \|\| b | true if a or b or both are true and false otherwise |

- 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 a *truth table:*

| a | !a |
|---|---|
| false | true |
| true | false |

# 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

# Logical AND and Logical OR

- A truth table shows all possible true-false combinations of the terms

- Since `&&` and `||` each have two operands, there are four possible combinations

| a | b | a && b | a \|\| b |
|---|---|---|---|
| false | false | false | false |
| false | true | false | true |
| true | false | false | true |
| true | true | true | true |

# Logical Operators

- Expressions that use logical operators can form complex conditions

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

- All logical operators have lower precedence than the relational operators

- Logical NOT has higher precedence than logical AND and logical OR

# Logical Operators

- Specific expressions can be evaluated using truth tables:

| done | count > MAX | !done | !done && (count > MAX) |
|---|---|---|---|
| false | false | true | false |
| false | true | true | true |
| true | false | false | false |
| true | true | false | false |

# 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

# The if Statement

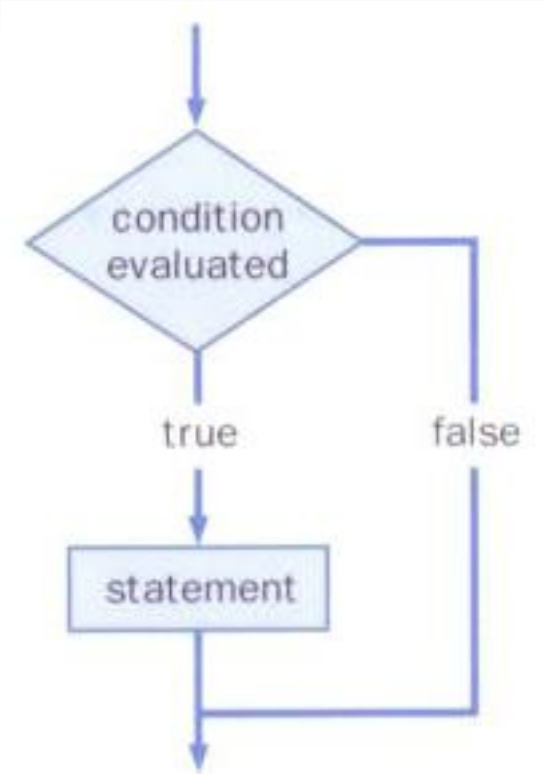
- Consider the following if statement:

```
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 executed -- if it isn't, it is skipped.
- Either way, the call to `println` is executed next

# The if Statement

- The logic of an if statement:



```java
//*******************************************************************
//  Age.java       Java Foundations
//
//  Demonstrates the use of an if statement.
//*******************************************************************

import java.util.Scanner;

public class Age
{
   //-----------------------------------------------------------------
   //  Reads the user's age and prints comments accordingly.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      final int MINOR = 21;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter your age: ");
      int age = scan.nextInt();

      System.out.println("You entered: " + age);

      if (age < MINOR)
         System.out.println("Youth is a wonderful thing. Enjoy.");

      System.out.println("Age is a state of mind.");
   }
}
```

# Indentation

- The statement controlled by the `if` statement is indented to indicate that relationship
- The use of a consistent indentation style makes a program easier to read and understand
- Although it makes no difference to the compiler, proper indentation is crucial

"Always code as if the person who ends up maintaining your code will be a violent psychopath who knows where you live."

-- Martin Golding

# 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

```
//*******************************************************************
//  Wages.java       Java Foundations
//
//  Demonstrates the use of an if-else statement.
//*******************************************************************

import java.text.NumberFormat;
import java.util.Scanner;

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

      Scanner scan = new Scanner(System.in);

      double pay = 0.0;

      System.out.print("Enter the number of hours worked: ");
      int hours = scan.nextInt();
```

```
      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));
   }
}
```

# Block Statements

- Several statements can be grouped together into a *block statement* delimited by braces

- A block statement can be used wherever a statement is called for in the Java syntax rules

```
if (total > MAX)
{
   System.out.println("Error!!");
   errorCount++;
}
```

# The if-else Statement

**If Statement**

boolean condition

executed if the condition is true

```
if (total <= cash)
    cash = cash - total;
else
{
    system.out.println("Insufficient cash.");
    total = 0
}
```

block statement

executed if the condition is false

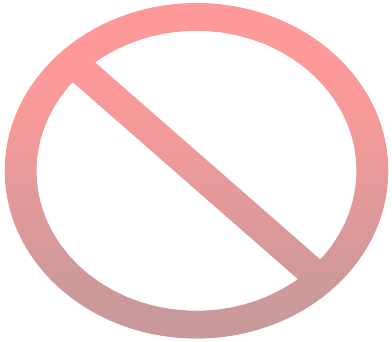
```
//*******************************************************************
//  Guessing.java       Java Foundations
//
//  Demonstrates the use of a block statement in an if-else.
//*******************************************************************

import java.util.*;

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

      Scanner scan = new Scanner(System.in);
      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 = scan.nextInt();

      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);
      }
   }
}
```

# Indentation Revisited

- Remember that indentation is for the human reader, and is ignored by the computer

```
if (total > MAX)
   System.out.println("Error!!");
   errorCount++;
```



- Despite what is implied by the indentation, the increment will occur whether the condition is true or not

# The if-else Statement

- In an `if-else` statement, the `if` portion, or the `else` portion, or both, could be block statements

```
if (total > MAX)
{
   System.out.println("Error!!");
   errorCount++;
}
else
{
   System.out.println("Total: " + total);
   current = total*2;
}
```

# The Conditional Operator

- Java has a *conditional operator* that uses a boolean condition to determine which of two expressions is evaluated
- Its syntax is

  ***condition*** `?` ***expression1*** `:` ***expression2***

- If the ***condition*** is true, ***expression1*** is evaluated; if it is false, ***expression2*** is evaluated
- The value of the entire conditional operator is the value of the selected expression

# The Conditional Operator

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

- For example

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

- If `num1` is greater than `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

# 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

```java
//*******************************************************************
//  MinOfThree.java       Java Foundations
//
//  Demonstrates the use of nested if statements.
//*******************************************************************

import java.util.Scanner;

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;

      Scanner scan = new Scanner(System.in);

      System.out.println("Enter three integers: ");
      num1 = scan.nextInt();
      num2 = scan.nextInt();
      num3 = scan.nextInt();
```

```
      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);
   }
}
```

# Comparing Data

- When comparing data using boolean expressions, it's important to understand the nuances of certain data types

- Let's examine some key situations:
  - comparing floating point values for equality
  - comparing characters
  - comparing strings (alphabetical order)
  - comparing object vs. comparing object references

# Comparing Float Values

- You should rarely use the equality operator (`==`) when comparing two floating point values (`float` or `double`)
- Two floating point values are equal only if their underlying binary representations match exactly
- Computations often result in slight differences that may be irrelevant
- In many situations, you might consider two floating point numbers to be “close enough” even if they aren't exactly equal

# Comparing Float Values

- To determine the equality of two floats, you may want to use the following technique:

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

- If the difference between the two floating point values is less than the tolerance, they are considered to be equal
- The tolerance could be set to any appropriate level, such as 0.000001

# Comparing Characters

- As we've discussed, Java character data is based on the Unicode character set
- Unicode establishes a particular numeric value for each character, and therefore an ordering
- We can use relational operators on character data based on this ordering
- For example, the character `'+'` is less than the character `'J'` because it comes before it in the Unicode character set
- Appendix C provides an overview of Unicode

# Comparing Characters

- In Unicode, the digit characters (0-9) are contiguous and in order
- Likewise, the uppercase letters (A-Z) and lowercase letters (a-z) are contiguous and in order

| Characters | Unicode Values |
|---|---|
| 0 – 9 | 48 through 57 |
| A – Z | 65 through 90 |
| a – z | 97 through 122 |

# Comparing Strings

- Remember that in Java a character string is an object
- The `equals` method can be called with strings to determine if two strings contain exactly the same characters in the same order
- The `equals` method returns a boolean result

```
if (name1.equals(name2))
   System.out.println("Same name");
```

# Comparing Strings

- We cannot use the relational operators to compare strings
- The `String` class contains a method called `compareTo` to determine if one string comes before another
- A call to `name1.compareTo(name2)`
  - returns zero if `name1` and `name2` are equal (contain the same characters)
  - returns a negative value if `name1` is less than `name2`
  - returns a positive value if `name1` is greater than `name2`

# Comparing Strings

```
if (name1.compareTo(name2) < 0)
   System.out.println(name1 + "comes first");
else
   if (name1.compareTo(name2) == 0)
      System.out.println("Same name");
   else
      System.out.println(name2 + "comes first");
```

- Because comparing characters and strings is based on a character set, it is called a *lexicographic ordering*

# Lexicographic Ordering

- Lexicographic ordering 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"`

# == vs. equals

- The `==` operator can be applied to objects – it returns true if the two references are aliases of each other
- The `equals` method is defined for all objects, and unless we redefine it when we write a class, it has the same semantics as the `==` operator
- It has been redefined in the `String` class to compare the characters in the two strings
- You can/should redefine the `equals` method to return true under whatever conditions are appropriate

# The switch Statement

- The *switch statement* provides another way 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 case value that matches

# The switch Statement

- The general syntax of a switch statement:

switch and case are reserved words

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

}
```

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 may be appropriate, but often we want to execute only the statements associated with one case

# The switch Statement

- An example of a switch statement:

```
switch (option)
{
   case 'A':
      aCount++;
      break;
   case 'B':
      bCount++;
      break;
   case 'C':
      cCount++;
      break;
}
```

# 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
- 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 integer (`byte`, `short`, `int`, `long`) or a `char`
- It cannot be a `boolean` value or a floating point value (`float` or `double`)
- The implicit boolean condition in a `switch` statement is equality
- You cannot perform relational checks with a `switch` statement

```
//*******************************************************************
//  GradeReport.java       Java Foundations
//
//  Demonstrates the use of a switch statement.
//*******************************************************************

import java.util.Scanner;

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

      Scanner scan = new Scanner(System.in);

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

      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.print("below average. Please see the ");
            System.out.println("instructor for assistance.");
            break;
         default:
            System.out.println("not passing.");
      }
   }
}
```

# Loops

- *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 Loop

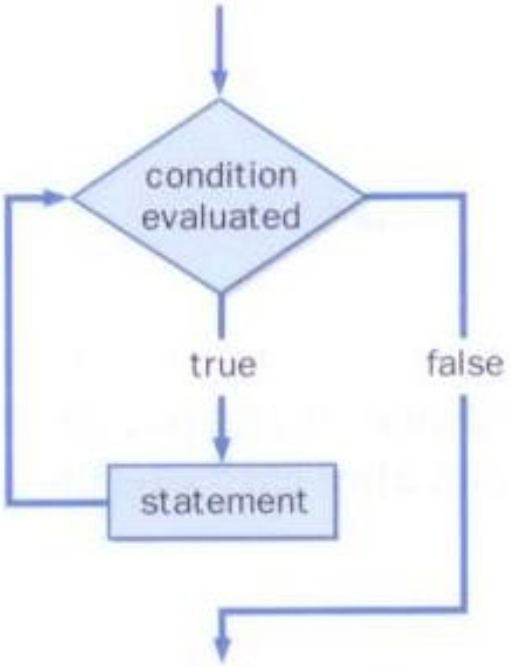
- A while loop has the following syntax

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

- If the condition is true, the statement is executed
- Then the condition is evaluated again, and if it is still true, the statement is executed again
- The statement is executed repeatedly until the condition becomes false

# The while Loop

- The logic of a while loop:



# The while Loop

- Example:

```
int count = 1;
while (count <= 5)
{
   System.out.println (count);
   count++;
}
```

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

# The while Loop

- Let's look at some examples of loop processing
- A loop can be used to maintain a *running sum*
- A *sentinel value* is a special input value that represents the end of input

```java
//*******************************************************************
//  Average.java       Java Foundations
//
//  Demonstrates the use of a while loop, a sentinel value, and a
//  running sum.
//*******************************************************************

import java.text.DecimalFormat;
import java.util.Scanner;

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;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter an integer (0 to quit): ");
      value = scan.nextInt();
```

```
      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 = scan.nextInt();
      }

      System.out.println();

      if (count == 0)
         System.out.println("No values were entered.");
      else
      {
         average = (double)sum / count;

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

# The while Loop

- A loop can also be used for *input validation*, making a program more *robust*

While-Loop

boolean condition

```
while = (input <= 0)
{
    System.out.println("Input must be positive.")
    input = scan.nextInt();
}
```

executed repeatedly as long as the condition is true

```java
//*******************************************************************
//  WinPercentage.java       Java Foundations
//
//  Demonstrates the use of a while loop for input validation.
//*******************************************************************

import java.text.NumberFormat;
import java.util.Scanner;

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;

      Scanner scan = new Scanner(System.in);

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

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

      ratio = (double)won / NUM_GAMES;

      NumberFormat fmt = NumberFormat.getPercentInstance();

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

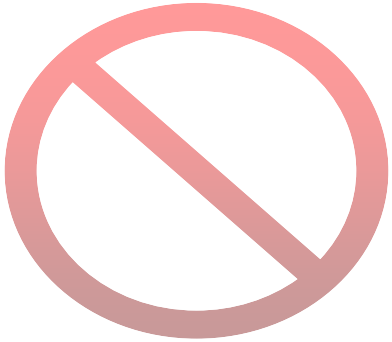
# Infinite Loops

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

# Infinite Loops

- An example of an infinite loop:

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



- This loop will continue executing until interrupted (Control-C) or until an underflow error occurs

# Nested Loops

- Similar to nested `if` statements, loops can be nested as well
- That is, the body of a loop can contain another loop
- For each iteration of the outer loop, the inner loop iterates completely

# Nested Loops

- How many times will the output be printed?

```
count1 = 1;
while (count1 <= 10)
{
   count2 = 1;
   while (count2 <= 50)
   {
      System.out.println ("Here again");
      count2++;
   }
   count1++;
}
```

```java
//*******************************************************************
//  PalindromeTester.java       Java Foundations
//
//  Demonstrates the use of nested while loops.
//*******************************************************************

import java.util.Scanner;

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

      Scanner scan = new Scanner(System.in);

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

         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 = scan.nextLine();
     }
   }
}
```

# Iterators

- An *iterator* is an object that allows you to process a collection of items one at a time
- It lets you step through each item in turn and process it as needed
- An iterator object has a `hasNext` method that returns true if there is at least one more item to process
- The `next` method returns the next item
- Iterator objects are defined using the `Iterator` interface, which is discussed further in Chapter 9

# Iterators

- Some classes in the Java API are iterators
- The `Scanner` class is an iterator
  - the `hasNext` method returns true if there is more data to be scanned
  - the `next` method returns the next scanned token as a string
- The `Scanner` class also has variations on the `hasNext` method for specific data types (such as `hasNextInt`)

# Iterators

- The fact that a `Scanner` is an iterator is particularly helpful when reading input from a file
- Suppose we wanted to read and process a list of URLs stored in a file
- One scanner can be set up to read each line of the input until the end of the file is encountered
- Another scanner can be set up for each URL to process each part of the path

```
//*******************************************************************
//  URLDissector.java       Java Foundations
//
//  Demonstrates the use of Scanner to read file input and parse it
//  using alternative delimiters.
//*******************************************************************

import java.util.Scanner;
import java.io.*;

public class URLDissector
{
   //-----------------------------------------------------------------
   //  Reads urls from a file and prints their path components.
   //-----------------------------------------------------------------
   public static void main(String[] args) throws IOException
   {
      String url;
      Scanner fileScan, urlScan;

      fileScan = new Scanner(new File("websites.inp"));

      // Read and process each line of the file
      while (fileScan.hasNextLine())
      {
         url = fileScan.nextLine();
         System.out.println("URL: " + url);
```

```
        urlScan = new Scanner(url);
        urlScan.useDelimiter("/");

        //  Print each part of the url
        while (urlScan.hasNext())
           System.out.println("   " + urlScan.next());

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

# The do Loop

- The do loop has the following syntax:

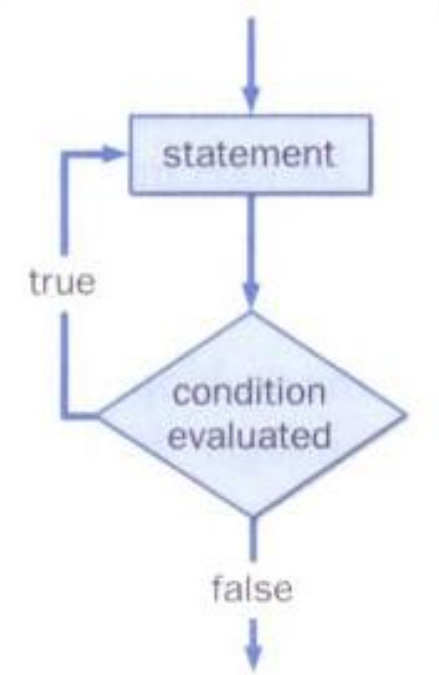
```
do
{
  statement;
}
while ( condition )
```

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

- The statement is executed repeatedly until the condition becomes false

# The do Loop

- The logic of a do loop:



# The do Loop

- An example of a do loop:

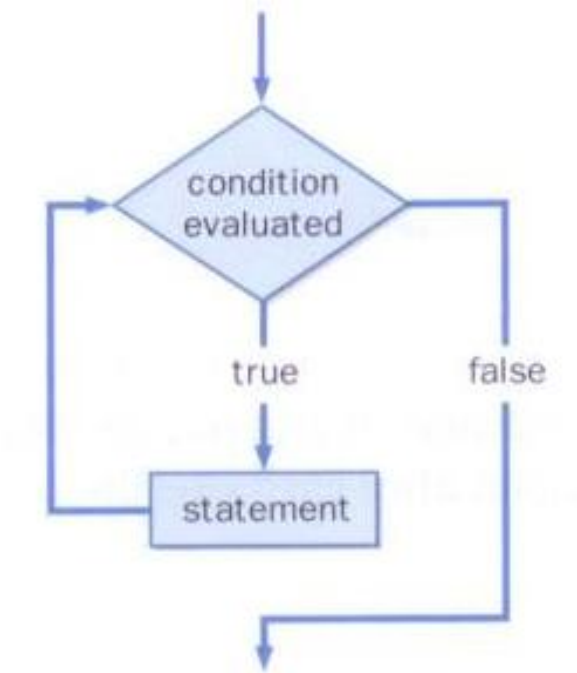
```
int count = 0;
do
{
   count++;
   System.out.println (count);
} while (count < 5);
```

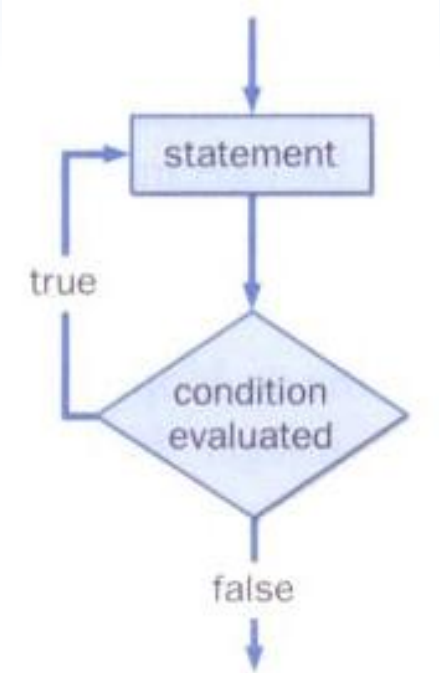
- The body of a do loop is executed at least once

```java
//*******************************************************************
//  ReverseNumber.java       Java Foundations
//
//  Demonstrates the use of a do loop.
//*******************************************************************

import java.util.Scanner;

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

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter a positive integer: ");
      number = scan.nextInt();
```

```
      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 Loops



# The for Loop

- The for loop has the following syntax:

The *initialization* is executed once before the loop begins

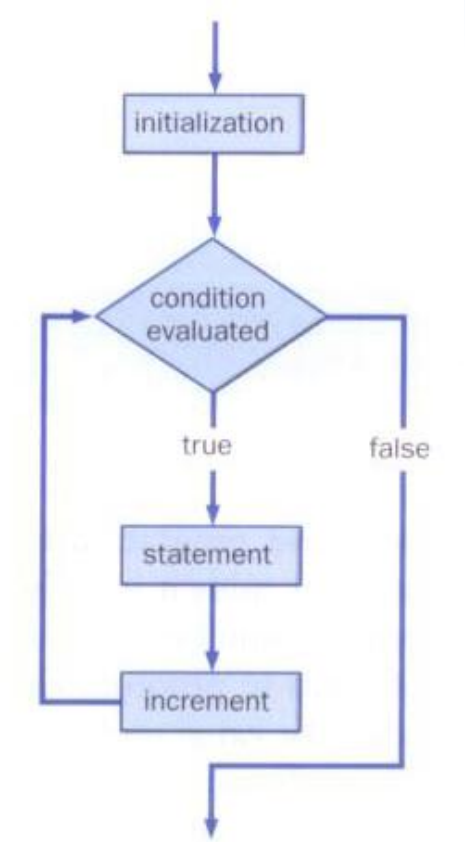
The *statement* is executed until the *condition* becomes false

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

The *increment* portion is executed at the end of each iteration

# The for Loop

- The logic of a for loop:



# The for Loop

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

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

# The for Loop

- An example of a for loop:

```
for (int count=1; count <= 5; count++)
   System.out.println (count);
```

- The initialization section can be used to declare a variable
- Like a while loop, the condition of a for loop is tested prior to executing the loop body
- Therefore, the body of a for loop will execute zero or more times

# The for Loop

- The increment section can perform any calculation

```
for (int num=100; num > 0; num -= 5)
   System.out.println (num);
```

- A for loop is well suited for executing statements a specific number of times that can be calculated or determined in advance

```java
//*******************************************************************
//  Multiples.java       Java Foundations
//
//  Demonstrates the use of a for loop.
//*******************************************************************

import java.util.Scanner;

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;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter a positive value: ");
      value = scan.nextInt();

      System.out.print("Enter an upper limit: ");
      limit = scan.nextInt();
```

```
      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();
      }
   }
}
```

```java
//*******************************************************************
//  Stars.java       Java Foundations
//
//  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 Loop

- 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

# Iterators and for Loops

- A variant of the `for` loop simplifies the repetitive processing for any object that implements the `Iterable` interface
- An `Iterable` interface provides an iterator
- For example, if `BookList` is an `Iterable` object that manages `Book` objects, the following loop will print each book:

```
for (Book myBook : BookList)
   System.out.println (myBook);
```

# The for-each Loop

- This style of `for` loop can be read "for each `Book` in `BookList`, …"
- This version is sometimes referred to as the *for-each* loop
- It eliminates the need to call the `hasNext` and `next` methods explicitly
- It also will be helpful when processing arrays, which are discussed in Chapter 7