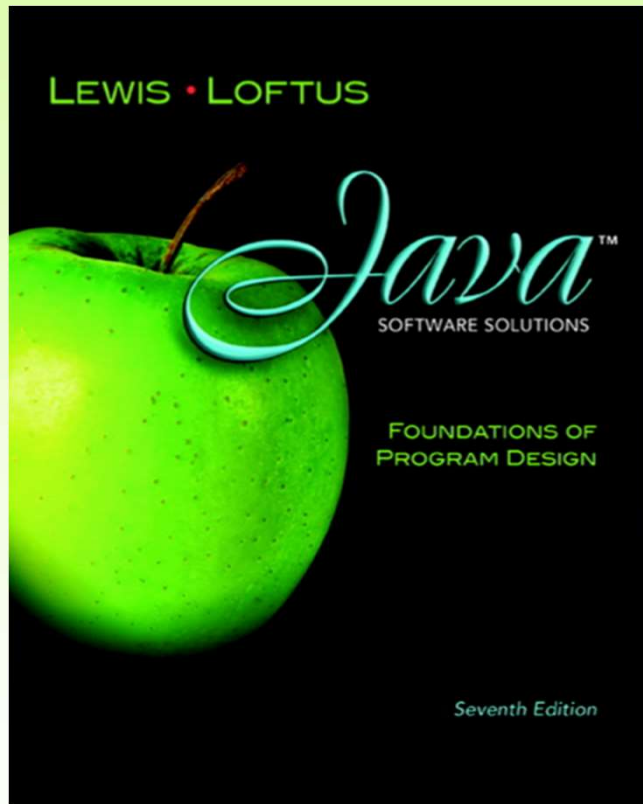# Chapter 5
Conditionals and Loops

Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus

Conditionals and Loops

- Now we will examine programming statements that allow us to:

  - make decisions
  - repeat processing steps in a loop

- Chapter 5 focuses on:

  - boolean expressions
  - the if and if-else statements
  - comparing data
  - while loops
  - iterators
  - more drawing techniques
  - more GUI components

Outline

→ **Boolean Expressions**

The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources

Check Boxes and Radio Buttons

Flow of Control

- Unless specified otherwise, the order of statement execution through a method is linear: one after another

- Some programming statements allow us to make decisions and perform repetitions

- These decisions are based on *boolean expressions* (also called *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

- They are sometimes called *selection statements*

- Conditional statements give us the power to make basic decisions

- The Java conditional statements are the:
  - if and if-else statement
  - switch statement

- We'll explore the switch statement in Chapter 6

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 (=)

Boolean Expressions

- An `if` statement with its boolean condition:

```
if (sum > MAX)
    delta = sum - MAX;
```

- 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

- See `Age.java`

```java
//***********************************************************
//  Age.java        Author: Lewis/Loftus
//
//  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();
```

continue

continue

```
    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.");
    }
}
```

continue

```
        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.");
    }
}
```

Sample Run

```
Enter your age: 47
You entered: 47
Age is a state of mind.
```

Another Sample Run

```
Enter your age: 12
You entered: 12
Youth is a wonderful thing. Enjoy.
Age is a state of mind.
```

Logical Operators

- Boolean expressions can also 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 a *truth table*:

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

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 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

- 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

- The ! operator has higher precedence than && and ||

Boolean Expressions

- 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

Short-Circuited Operators

- The processing of `&&` and `||` 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 should be used carefully

Outline

Boolean Expressions

➡ **The `if` Statement**

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class
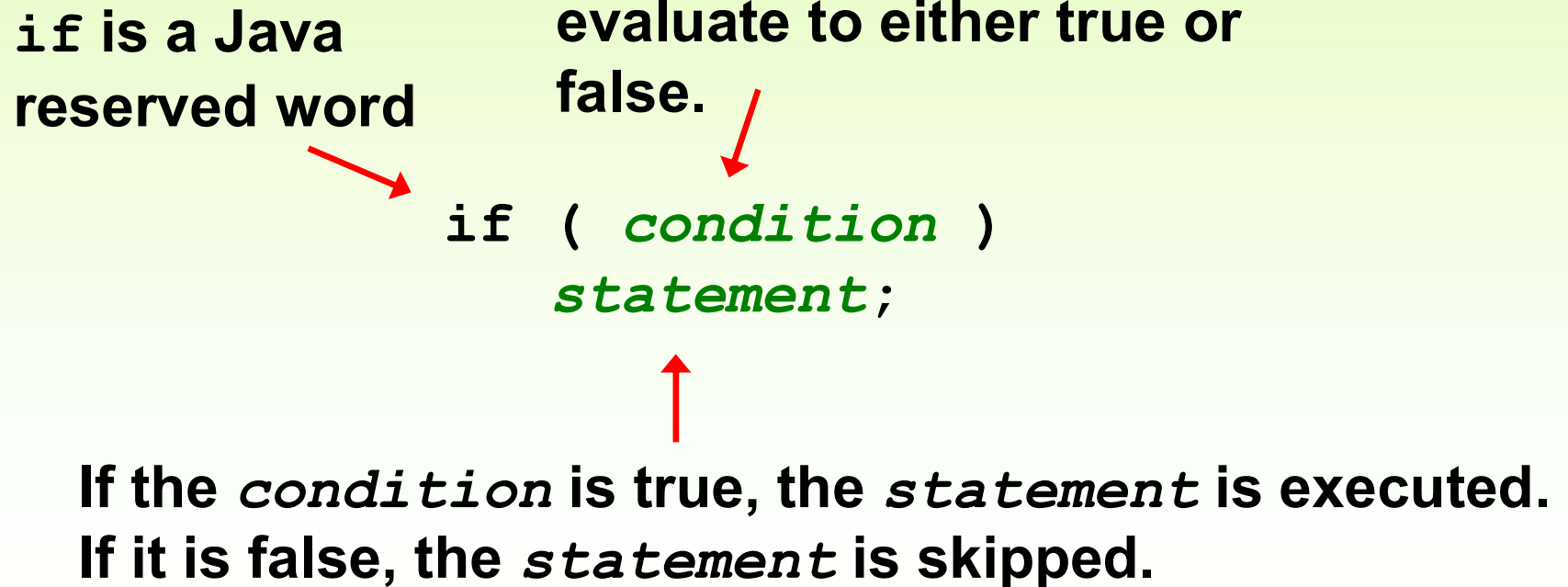
Determining Event Sources

Check Boxes and Radio Buttons

The if Statement

- Let's now look at the `if` statement in more detail

- The *if statement* has the following syntax:

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

`if` is a Java reserved word
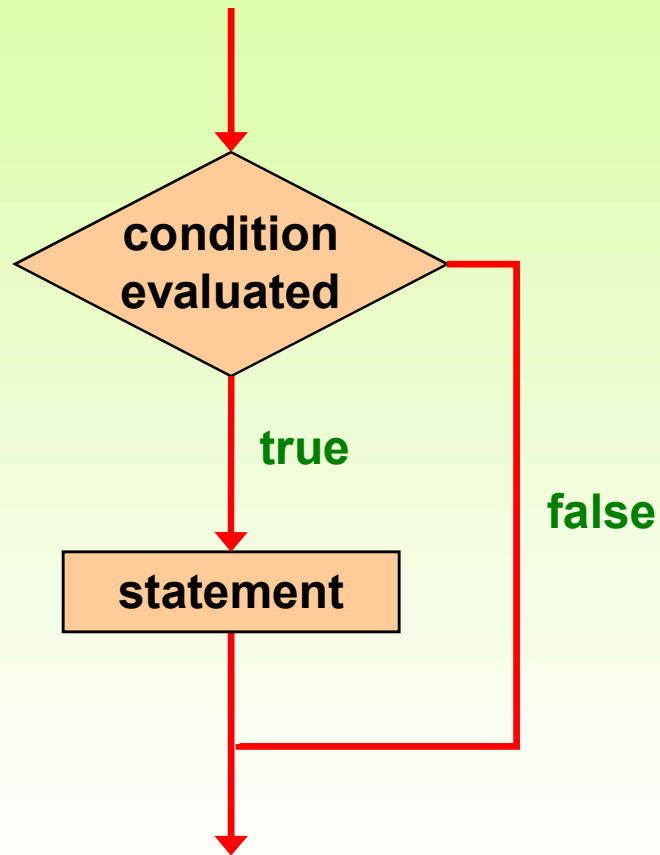
```
if ( condition )
    statement;
```

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

Logic of an if statement

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

- The compiler ignores indentation, which can lead to errors if the indentation is not correct

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

-- Martin Golding

Quick Check

What do the following statements do?

```
if (total != stock + warehouse)
    inventoryError = true;




if (found || !done)
    System.out.println("Ok");
```

Quick Check

What do the following statements do?

```
if (total != stock + warehouse)
    inventoryError = true;
```

Sets the boolean variable to true if the value of `total` is not equal to the sum of `stock` and `warehouse`

```
if (found || !done)
    System.out.println("Ok");
```

Prints "Ok" if `found` is true or `done` is false

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

- See `Wages.java`

```java
//*************************************************************
//  Wages.java         Author: Lewis/Loftus
//
//  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;

continue
```

continue

```java
        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));
    }
}
```

continue

```
        System.o                              ");
        int hou

        System.o

        // 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));
    }
}
```

Sample Run

Enter the number of hours worked: **46**

Gross earnings: $404.25

Logic of an if-else statement
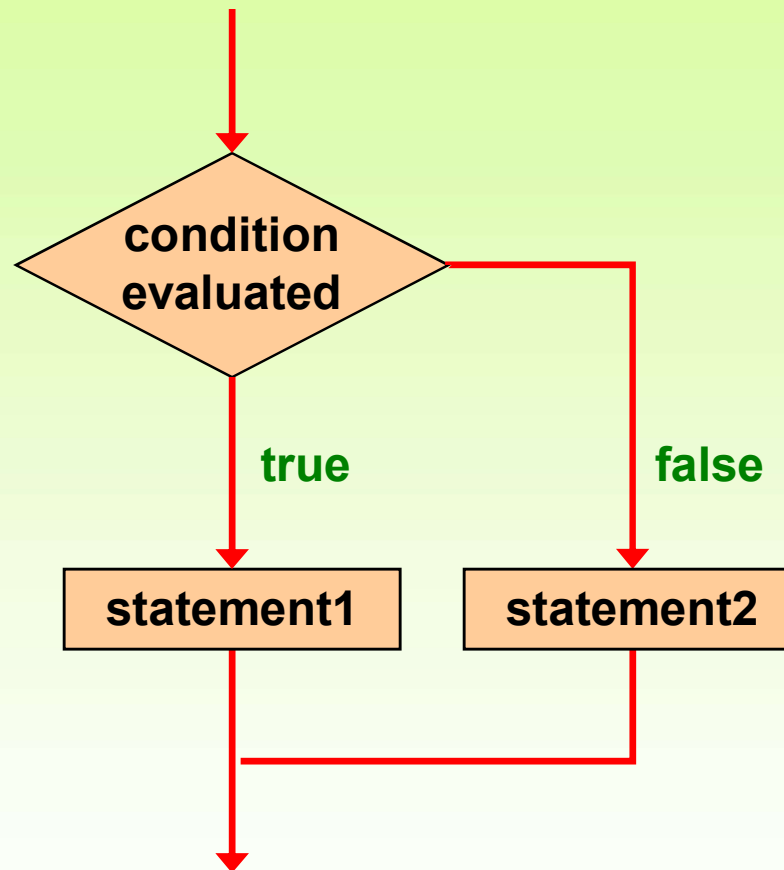
The Coin Class

- Let's look at an example that uses a class that represents a coin that can be flipped

- Instance data is used to indicate which face (heads or tails) is currently showing

- See `CoinFlip.java`
- See `Coin.java`

```java
//*************************************************************
//  CoinFlip.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an if-else statement.
//*************************************************************

public class CoinFlip
{
   //----------------------------------------------------------
   //  Creates a Coin object, flips it, and prints the results.
   //----------------------------------------------------------
   public static void main (String[] args)
   {
      Coin myCoin = new Coin();

      myCoin.flip();

      System.out.println (myCoin);

      if (myCoin.isHeads())
         System.out.println ("You win.");
      else
         System.out.println ("Better luck next time.");
   }
}
```

```
//********************************               *********************
//   CoinFlip.java                 Sample Run
//                                  Tails
//   Demonstrates the              Better luck next time.         *********************
//********************************               *********************

public class CoinFlip
{
   //----------------------------------------------------------------
   //   Creates a Coin object, flips it, and prints the results.
   //----------------------------------------------------------------
   public static void main (String[] args)
   {
      Coin myCoin = new Coin();

      myCoin.flip();

      System.out.println (myCoin);

      if (myCoin.isHeads())
         System.out.println ("You win.");
      else
         System.out.println ("Better luck next time.");
   }
}
```

```java
//************************************************************
//  Coin.java       Author: Lewis/Loftus
//
//  Represents a coin with two sides that can be flipped.
//************************************************************

public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;

   private int face;

   //---------------------------------------------------------
   //  Sets up the coin by flipping it initially.
   //---------------------------------------------------------
   public Coin ()
   {
      flip();
   }
```

continue

continue

```java
//---------------------------------------------------------
//  Flips the coin by randomly choosing a face value.
//---------------------------------------------------------
public void flip ()
{
   face = (int) (Math.random() * 2);
}


//---------------------------------------------------------
//  Returns true if the current face of the coin is heads.
//---------------------------------------------------------
public boolean isHeads ()
{
   return (face == HEADS);
}
```

continue

continue

```java
   //------------------------------------------------------------
   //  Returns the current face of the coin as a string.
   //------------------------------------------------------------
   public String toString()
   {
      String faceName;

      if (face == HEADS)
         faceName = "Heads";
      else
         faceName = "Tails";

      return faceName;
   }
}
```

Indentation Revisited

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

```
if (depth >= UPPER_LIMIT)
    delta = 100;
else
    System.out.println("Reseting Delta");
    delta = 0;
```

- Despite what the indentation implies, `delta` will be set to 0 no matter what

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++;
}
```

Block Statements

- The `if` clause, or the `else` clause, or both, could govern block statements

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

- See `Guessing.java`

```
//***********************************************************
//  Guessing.java       Author: Lewis/Loftus
//
//  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;

continue
```

continue

```java
    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);
    }
    }
}
```

Sample Run

```
I'm thinking of a number between 1 and 10. Guess what it is: 6
That is not correct, sorry.
The number was 9
```

```java
        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` 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

- See `MinOfThree.java`

```
//**********************************************************************
//  MinOfThree.java        Author: Lewis/Loftus
//
//  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();
```

continue

continue

```java
    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);
    }
}
```

continue

```
    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);
    }
}
```

Outline

Boolean Expressions

The `if` Statement

⟹ **Comparing Data**

The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources

Check Boxes and Radio Buttons

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, 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 the `compareTo` method for determining 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

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

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

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"`

Comparing Objects

- 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, but 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

- When you write a class, you can redefine the `equals` method to return true under whatever conditions are appropriate

Outline

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: `while`, `do`, and `for` loops

- The `do` and `for` loops are discussed in Chapter 6

The while Statement

- A *while statement* 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

Logic of a while Loop
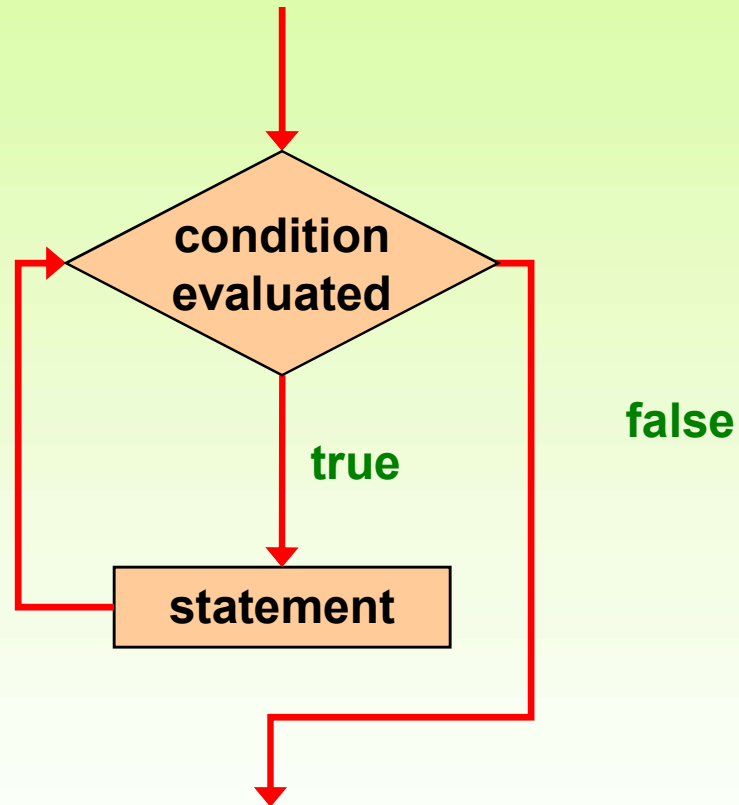
The while Statement

- An example of a while statement:

```
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

Sentinel Values

- 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

- See `Average.java`

```java
//************************************************************
//   Average.java        Author: Lewis/Loftus
//
//   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();

continue
```

continue

```java
    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();
    }
```

continue

continue

```java
    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));
    }
  }
}
```

continue

```
        System.ou
    
    if (count
        System
    else
    {
        average
        
        Decimal
        System                          at(average));
    }
    }
}
```

Sample Run

```
Enter an integer (0 to quit): 25
The sum so far is 25
Enter an integer (0 to quit): 164
The sum so far is 189
Enter an integer (0 to quit): -14
The sum so far is 175
Enter an integer (0 to quit): 84
The sum so far is 259
Enter an integer (0 to quit): 12
The sum so far is 271
Enter an integer (0 to quit): -35
The sum so far is 236
Enter an integer (0 to quit): 0

The average is 39.333
```

Input Validation

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

- It's generally a good idea to verify that input is valid (in whatever sense) when possible

- See `WinPercentage.java`

```
//***********************************************************
//  WinPercentage.java        Author: Lewis/Loftus
//
//  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();
```

continue

continue

```java
        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));
    }
}
```

continue

```
        while
        {
            S
            w
        }

        ratio = (double)won / NUM_GAMES;

        NumberFormat fmt = NumberFormat.getPercentInstance();

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

Sample Run

```
Enter the number of games won (0 to 12): -5
Invalid input. Please reenter: 13
Invalid input. Please reenter: 7

Winning percentage: 58%
```

Infinite Loops

- The body of a `while` 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 always 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

- See `PalindromeTester.java`

```java
//************************************************************
//  PalindromeTester.java       Author: Lewis/Loftus
//
//  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;
```

continue

continue

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

continue

```
        while                              & left < right)
        {
            lef
            rig
        }

        System

        if (le
            Sys                    alindrome.");
        else
            Sys                    drome.");

        System
        System                     n)? ");
        anothe
        }
    }
}
```

Sample Run

```
Enter a potential palindrome:
radar

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
able was I ere I saw elba

That string IS a palindrome.

Test another palindrome (y/n)? y
Enter a potential palindrome:
abracadabra

That string is NOT a palindrome.

Test another palindrome (y/n)? n
```

Quick Check

How many times will the string "Here" be printed?

```
count1 = 1;
while (count1 <= 10)
{
    count2 = 1;
    while (count2 < 20)
    {
        System.out.println ("Here");
        count2++;
    }
    count1++;
}
```

Quick Check

How many times will the string "Here" be printed?

```
count1 = 1;
while (count1 <= 10)
{
    count2 = 1;
    while (count2 < 20)
    {
        System.out.println ("Here");
        count2++;
    }
    count1++;
}
```

10 * 19 = 190

Outline
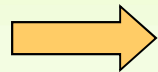
Iterators

- 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 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 7

Iterators

- Several classes in the Java standard class library 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

- See `URLDissector.java`

```java
//************************************************************
//   URLDissector.java        Author: Lewis/Loftus
//
//   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("urls.inp"));

continue
```

continue

```java
    // Read and process each line of the file
    while (fileScan.hasNext())
    {
        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();
    }
  }
}
```

```
continue

    // Rea
    while
    {
        url
        Sys

        url
        url

        //
        whi

        Sys
    }
}
}
```

Sample Run

```
URL: www.google.com
    www.google.com

URL: www.linux.org/info/gnu.html
    www.linux.org
    info
    gnu.html

URL: thelyric.com/calendar/
    thelyric.com
    calendar

URL: www.cs.vt.edu/undergraduate/about
    www.cs.vt.edu
    undergraduate
    about

URL: youtube.com/watch?v=EHCRimwRGLs
    youtube.com
    watch?v=EHCRimwRGLs
```

Outline

Boolean Expressions

The `if` Statement

Comparing Data

The `while` Statement

Iterators
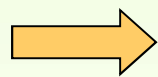
→ **The `ArrayList` Class**

Determining Event Sources

Check Boxes and Radio Buttons

The ArrayList Class

- An `ArrayList` object stores a list of objects, and is often processed using a loop

- The `ArrayList` class is part of the `java.util` package

- You can reference each object in the list using a numeric index

- An `ArrayList` object grows and shrinks as needed, adjusting its capacity as necessary

The ArrayList Class

- Index values of an `ArrayList` begin at 0 (not 1):

0	"Bashful"
1	"Sleepy"
2	"Happy"
3	"Dopey"
4	"Doc"

- Elements can be inserted and removed

- The indexes of the elements adjust accordingly

ArrayList Methods

- Some `ArrayList` methods:

```
boolean add (E obj)

void add (int index, E obj)

Object remove (int index)

Object get (int index)

boolean isEmpty()

int size()
```

The ArrayList Class

- The type of object stored in the list is established when the `ArrayList` object is created:

  ```
  ArrayList<String> names = new ArrayList<String>();

  ArrayList<Book> list = new ArrayList<Book>();
  ```

- This makes use of Java *generics*, which provide additional type checking at compile time

- An `ArrayList` object cannot store primitive types, but that's what wrapper classes are for

- See `Beatles.java`

```
//***********************************************************
//   Beatles.java        Author: Lewis/Loftus
//
//   Demonstrates the use of a ArrayList object.
//***********************************************************

import java.util.ArrayList;

public class Beatles
{
   //--------------------------------------------------------
   //   Stores and modifies a list of band members.
   //--------------------------------------------------------
   public static void main (String[] args)
   {
      ArrayList<String> band = new ArrayList<String>();

      band.add ("Paul");
      band.add ("Pete");
      band.add ("John");
      band.add ("George");
```

continue
```

**continue**

```java
      System.out.println (band);
      int location = band.indexOf ("Pete");
      band.remove (location);

      System.out.println (band);
      System.out.println ("At index 1: " + band.get(1));
      band.add (2, "Ringo");

      System.out.println ("Size of the band: " + band.size());
      int index = 0;
      while (index < band.size())
      {
         System.out.println (band.get(index));
         index++;
      }
   }
}
```

continue

```
        System.out.p      [Paul, Pete, John, George]
        int location      [Paul, John, George]
        band.remove        At index 1: John
                           Size of the band: 4
        System.out.p       Paul
        System.out.p       John                         (1));
        band.add (2,       Ringo
                           George
        System.out.p                                    nd.size());
        int index =
        while (index < band.size())
        {
            System.out.println (band.get(index));
            index++;
        }
    }
}
```

Output

Outline

Boolean Expressions

The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

→ **Determining Event Sources**

Check Boxes and Radio Buttons

Determining Event Sources

- Recall that interactive GUIs require establishing a relationship between components and the listeners that respond to component events

- One listener object can be used to listen to two different components

- The source of the event can be determined by using the `getSource` method of the event passed to the listener

- See `LeftRight.java`
- See `LeftRightPanel.java`

```
//*****************************************************************
//  LeftRight.java       Authors: Lewis/Loftus
//
//  Demonstrates the use of one listener for multiple buttons.
//*****************************************************************

import javax.swing.JFrame;

public class LeftRight
{
    //--------------------------------------------------------------
    //  Creates the main program frame.
    //--------------------------------------------------------------
    public static void main (String[] args)
    {
        JFrame frame = new JFrame ("Left Right");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(new LeftRightPanel());

        frame.pack();
        frame.setVisible(true);
    }
}
```

```
//*********************              **********************
//  LeftRight.java
//
//  Demonstrates the u                          ple buttons.
//*********************              **********************

import javax.swing.JFr

public class LeftRight
{
   //------------------------------------------------------------
   //  Creates the main program frame.
   //------------------------------------------------------------
   public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Left Right");
      frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

      frame.getContentPane().add(new LeftRightPanel());

      frame.pack();
      frame.setVisible(true);
   }
}
```


Left Right
Left
Left   Right

```
//***********************************************************
//   LeftRightPanel.java        Authors: Lewis/Loftus
//
//   Demonstrates the use of one listener for multiple buttons.
//***********************************************************

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class LeftRightPanel extends JPanel
{
    private JButton left, right;
    private JLabel label;
    private JPanel buttonPanel;

continue
```

continue

```java
   //------------------------------------------------------------
   //  Constructor: Sets up the GUI.
   //------------------------------------------------------------
   public LeftRightPanel ()
   {
      left = new JButton ("Left");
      right = new JButton ("Right");

      ButtonListener listener = new ButtonListener();
      left.addActionListener (listener);
      right.addActionListener (listener);

      label = new JLabel ("Push a button");

      buttonPanel = new JPanel();
      buttonPanel.setPreferredSize (new Dimension(200, 40));
      buttonPanel.setBackground (Color.blue);
      buttonPanel.add (left);
      buttonPanel.add (right);

      setPreferredSize (new Dimension(200, 80));
      setBackground (Color.cyan);
      add (label);
      add (buttonPanel);
   }
```

continue

continue

```java
//*******************************************************************
//  Represents a listener for both buttons.
//*******************************************************************
private class ButtonListener implements ActionListener
{
   //------------------------------------------------------------
   //  Determines which button was pressed and sets the label
   //  text accordingly.
   //------------------------------------------------------------
   public void actionPerformed (ActionEvent event)
   {
      if (event.getSource() == left)
         label.setText("Left");
      else
         label.setText("Right");
   }
}
}
```

Outline

Boolean Expressions

The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources
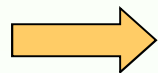
→ Check Boxes and Radio Buttons

Check Boxes

- A *check box* is a button that can be toggled on or off

- It is represented by the `JCheckBox` class

- Unlike a push button, which generates an action event, a check box generates an *item event* whenever it changes state

- The `ItemListener` interface is used to define item event listeners

- A check box calls the `itemStateChanged` method of the listener when it is toggled

Check Boxes

- Let's examine a program that uses check boxes to determine the style of a label's text string

- It uses the `Font` class, which embodies a character font's:

  - family name (such as Times or Courier)
  - style (bold, italic, or both)
  - font size

- See `StyleOptions.java`
- See `StyleOptionsPanel.java`

```java
//************************************************************
//  StyleOptions.java       Author: Lewis/Loftus
//
//  Demonstrates the use of check boxes.
//************************************************************

import javax.swing.JFrame;

public class StyleOptions
{
   //---------------------------------------------------------
   //  Creates and presents the program frame.
   //---------------------------------------------------------
   public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Style Options");
      frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

      StyleOptionsPanel panel = new StyleOptionsPanel();
      frame.getContentPane().add (panel);

      frame.pack();
      frame.setVisible(true);
   }
}
```

```java
//*****************                            ****************
//  StyleOptions.
//
//  Demonstrates
//*****************                            ****************

import javax.swin

public class StyleOptions
{
    //------------                              -------------
    //  Creates an
    //------------                              -------------
    public static
    {
        JFrame fram
        frame.setDe                        SE);

        StyleOption
        frame.getContentPane().add (panel);

        frame.pack();
        frame.setVisible(true);
    }
}
```

```
//***********************************************************************
//  StyleOptionsPanel.java      Author: Lewis/Loftus
//
//  Demonstrates the use of check boxes.
//***********************************************************************

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class StyleOptionsPanel extends JPanel
{
    private JLabel saying;
    private JCheckBox bold, italic;

continue
```

continue

```java
    //----------------------------------------------------------
    //  Sets up a panel with a label and some check boxes that
    //  control the style of the label's font.
    //----------------------------------------------------------
    public StyleOptionsPanel()
    {
        saying = new JLabel ("Say it with style!");
        saying.setFont (new Font ("Helvetica", Font.PLAIN, 36));

        bold = new JCheckBox ("Bold");
        bold.setBackground (Color.cyan);
        italic = new JCheckBox ("Italic");
        italic.setBackground (Color.cyan);

        StyleListener listener = new StyleListener();
        bold.addItemListener (listener);
        italic.addItemListener (listener);

        add (saying);
        add (bold);
        add (italic);

        setBackground (Color.cyan);
        setPreferredSize (new Dimension(300, 100));
    }
```

continue

continue

```java
//***************************************************************
//   Represents the listener for both check boxes.
//***************************************************************
private class StyleListener implements ItemListener
{
   //------------------------------------------------------------
   //   Updates the style of the label font style.
   //------------------------------------------------------------
   public void itemStateChanged (ItemEvent event)
   {
      int style = Font.PLAIN;

      if (bold.isSelected())
         style = Font.BOLD;

      if (italic.isSelected())
         style += Font.ITALIC;

      saying.setFont (new Font ("Helvetica", style, 36));
   }
}
}
```

Radio Buttons

- A group of *radio buttons* represents a set of mutually exclusive options – only one can be selected at any given time

- When a radio button from a group is selected, the button that is currently "on" in the group is automatically toggled off

- To define the group of radio buttons that will work together, each radio button is added to a `ButtonGroup` object

- A radio button generates an action event

Radio Buttons

- Let's look at a program that uses radio buttons to determine which line of text to display

- See `QuoteOptions.java`
- See `QuoteOptionsPanel.java`

```java
//************************************************************
//  QuoteOptions.java        Author: Lewis/Loftus
//
//  Demonstrates the use of radio buttons.
//************************************************************

import javax.swing.JFrame;

public class QuoteOptions
{
   //---------------------------------------------------------
   //  Creates and presents the program frame.
   //---------------------------------------------------------
   public static void main (String[] args)
   {
      JFrame frame = new JFrame ("Quote Options");
      frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

      QuoteOptionsPanel panel = new QuoteOptionsPanel();
      frame.getContentPane().add (panel);

      frame.pack();
      frame.setVisible(true);
   }
}
```

```
//*****************                          ***************
//  QuoteOptions.
//
//  Demonstrates
//*****************                          ***************

import javax.swin

public class QuoteOptions
{
    //---------------                          --------------
    //  Creates an
    //---------------                          --------------
    public static
    {
        JFrame fram
        frame.setDe                            SE);

        QuoteOption
        frame.getContentPane().add (panel);

        frame.pack();
        frame.setVisible(true);
    }
}
```
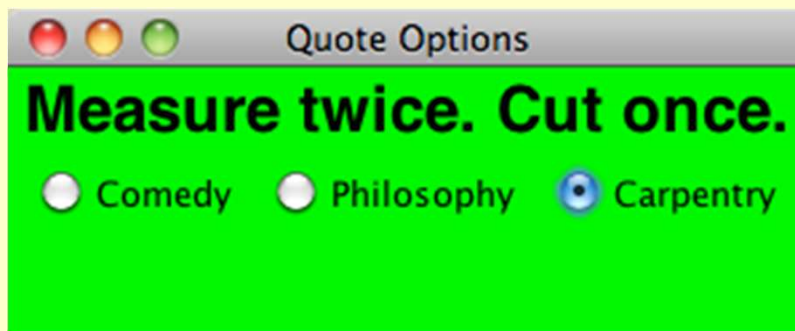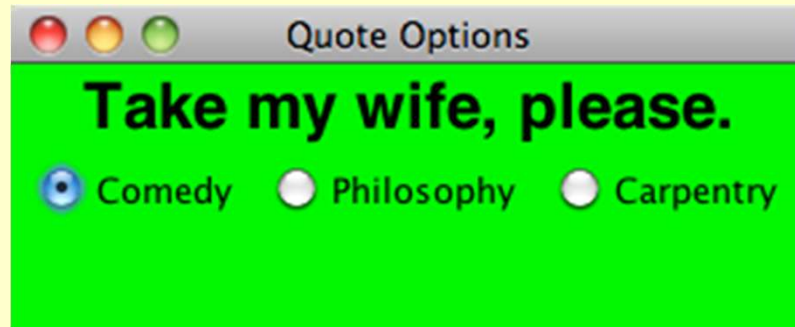
```java
//************************************************************
//   QuoteOptionsPanel.java       Author: Lewis/Loftus
//
//   Demonstrates the use of radio buttons.
//************************************************************

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class QuoteOptionsPanel extends JPanel
{
    private JLabel quote;
    private JRadioButton comedy, philosophy, carpentry;
    private String comedyQuote, philosophyQuote, carpentryQuote;

    //-----------------------------------------------------------
    //  Sets up a panel with a label and a set of radio buttons
    //   that control its text.
    //-----------------------------------------------------------
    public QuoteOptionsPanel()
    {
        comedyQuote = "Take my wife, please.";
        philosophyQuote = "I think, therefore I am.";
        carpentryQuote = "Measure twice. Cut once.";

        quote = new JLabel (comedyQuote);
        quote.setFont (new Font ("Helvetica", Font.BOLD, 24));
```

continue

continue

```
    comedy = new JRadioButton ("Comedy", true);
    comedy.setBackground (Color.green);
    philosophy = new JRadioButton ("Philosophy");
    philosophy.setBackground (Color.green);
    carpentry = new JRadioButton ("Carpentry");
    carpentry.setBackground (Color.green);

    ButtonGroup group = new ButtonGroup();
    group.add (comedy);
    group.add (philosophy);
    group.add (carpentry);

    QuoteListener listener = new QuoteListener();
    comedy.addActionListener (listener);
    philosophy.addActionListener (listener);
    carpentry.addActionListener (listener);

    add (quote);
    add (comedy);
    add (philosophy);
    add (carpentry);

    setBackground (Color.green);
    setPreferredSize (new Dimension(300, 100));
}
```

continue

continue

```java
//****************************************************************
//   Represents the listener for all radio buttons
//****************************************************************
private class QuoteListener implements ActionListener
{
   //---------------------------------------------------------
   //   Sets the text of the label depending on which radio
   //   button was pressed.
   //---------------------------------------------------------
   public void actionPerformed (ActionEvent event)
   {
      Object source = event.getSource();

      if (source == comedy)
         quote.setText (comedyQuote);
      else
         if (source == philosophy)
            quote.setText (philosophyQuote);
         else
            quote.setText (carpentryQuote);
   }
}
}
```

Summary

- Chapter 5 focused on:

    - boolean expressions
    - the if and if-else statements
    - comparing data
    - while loops
    - iterators
    - more drawing techniques
    - more GUI components