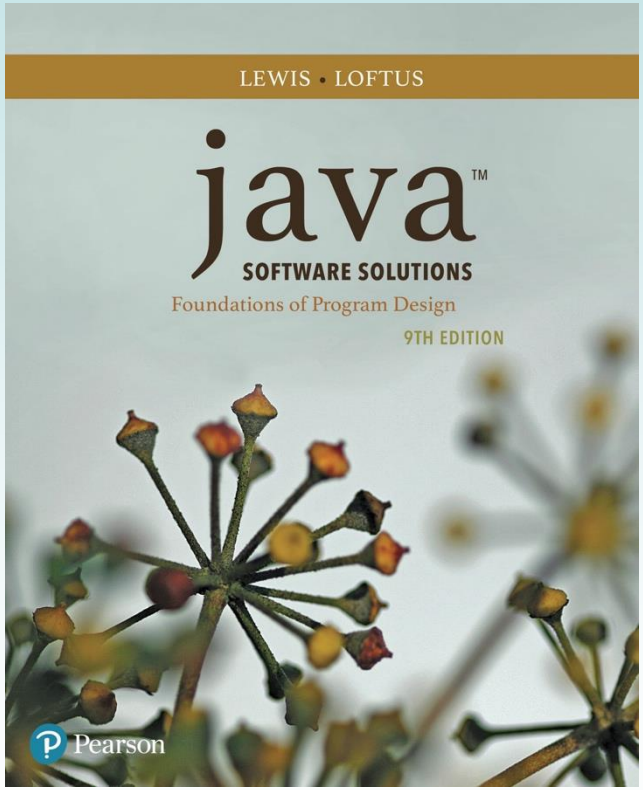
Chapter 5
Conditionals and Loops



Java Software Solutions

Foundations of Program Design

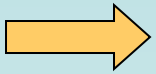
9th Edition

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
  - the `ArrayList` class
  - more GUI controls

Outline



Boolean Expressions

The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources

Managing Fonts

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`

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 `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

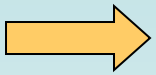
Short-Circuited Operators

- 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

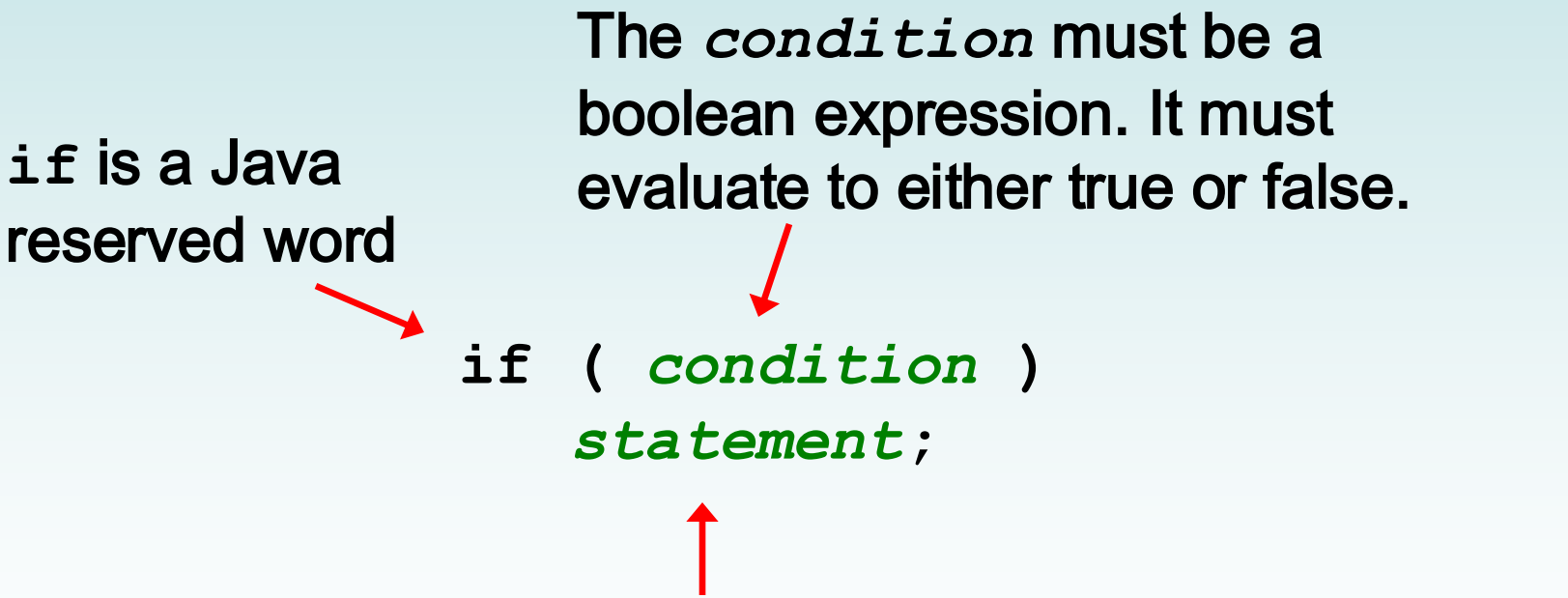
Determining Event Sources

Managing Fonts

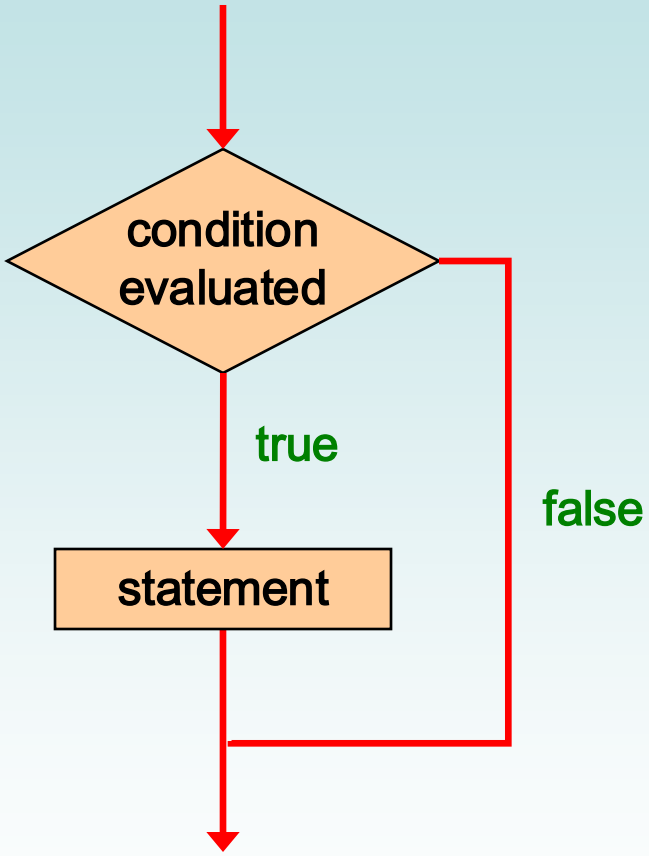
Check Boxes and Radio Buttons

The if Statement

- Let's now look at the `if` statement in more detail
- The *if statement* has the following syntax:



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

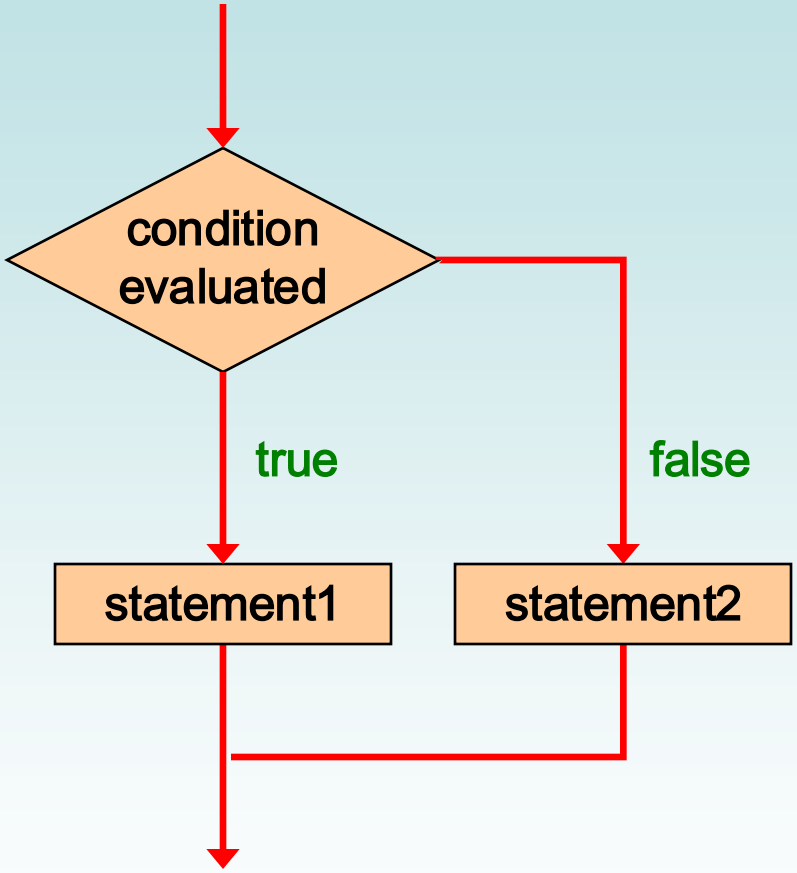
The if-else Statement

- 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`

Logic of an if-else statement



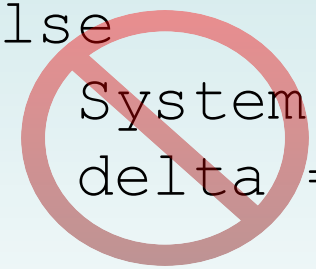
The Coin Class

- 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`

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

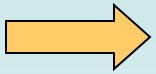
```
if (total > MAX)
{
   System.out.println("Error!!");
   errorCount++;
}
else
{
   System.out.println("Total: " + total);
   current = total*2;
}
```

- See `Guessing.java`

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`

Outline



Boolean Expressions

The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources

Managing Fonts

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.compareTo(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

Boolean Expressions

The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources

Managing Fonts

Check Boxes and Radio Buttons

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

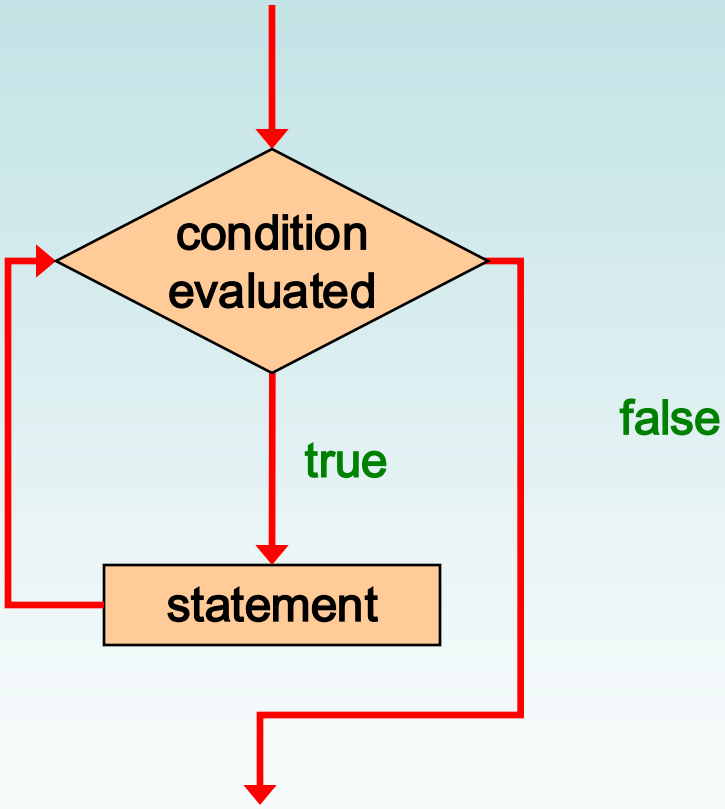
The while Statement

- 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`

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`

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`

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

Boolean Expressions

The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources

Managing Fonts

Check Boxes and Radio Buttons

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`

Outline

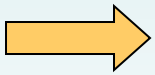
Boolean Expressions

The `if` Statement

Comparing Data

The `while` Statement

Iterators



The `ArrayList` Class

Determining Event Sources

Managing Fonts

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<E>` methods:

```
boolean add(E obj) //returns true

void add(int index, E obj)

E remove(int index)

E 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`

Outline

Boolean Expressions

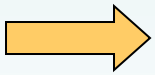
The `if` Statement

Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class



Determining Event Sources

Managing Fonts

Check Boxes and Radio Buttons

Determining Event Sources

- Recall that you must establish a relationship between controls and the event handlers that respond to events
- When appropriate, one event handler object can be used to listen to multiple controls
- The source of the event can be determined by using the `getSource` method of the event passed to the event handler
- See `RedOrBlue.java`

Outline

Boolean Expressions

The `if` Statement

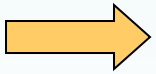
Comparing Data

The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources



Managing Fonts

Check Boxes and Radio Buttons

Managing Fonts

- The `Font` class represents a character font, which specify what characters look like when displayed
- A font can be applied to a `Text` object or any control that displays text (such as a `Button` or `Label`)
- A font is specifies:
  - *font family* (Arial, Courier, Helvetica)
  - *font size* (in units called points)
  - *font weight* (boldness)
  - *font posture* (italic or normal)

Managing Fonts

- A `Font` object is created using either the `Font` constructor or by calling the static `font` method
- The `Font` constructor can only take a font size, or a font family and size
- To set the font weight or font posture, use the `font` method, which can specify various combinations of font characteristics
- See `FontDemo.java`

Managing Fonts

- Note that setting the text color is not a function of the font applied
- It's set through the `Text` object directly
- The same is true for underlined text (or a "strike through" effect)

Outline

Boolean Expressions

The `if` Statement

Comparing Data

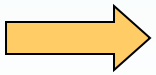
The `while` Statement

Iterators

The `ArrayList` Class

Determining Event Sources

Managing Fonts



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 JavaFX `CheckBox` class
- Checking or unchecking a check box produces an action event
- See `StyleOptions.java`
- See `StyleOptionsPane.java`

Check Boxes

- The `StyleOptionsPane` class uses two layout panes: `HBox` and `VBox`
- The `HBox` pane arranges its nodes into a single row horizontally
- The `VBox` pane arranges its nodes into a single column vertically
- `StyleOptionsPane` extends `VBox`, and is used to put the text above the check boxes
- The `HBox` puts the check boxes side by side

Check Boxes

- The event handler method is called when either check box is toggled
- Instead of tracking which box was changed, the method just checks the current status of both boxes and sets the font accordingly

Radio Buttons

- Let's look at a similar example that uses *radio buttons*
- A group of radio buttons represents a set of mutually exclusive options – only one button 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
- See `QuoteOptions.java`
- See `QuoteOptionsPane.java`

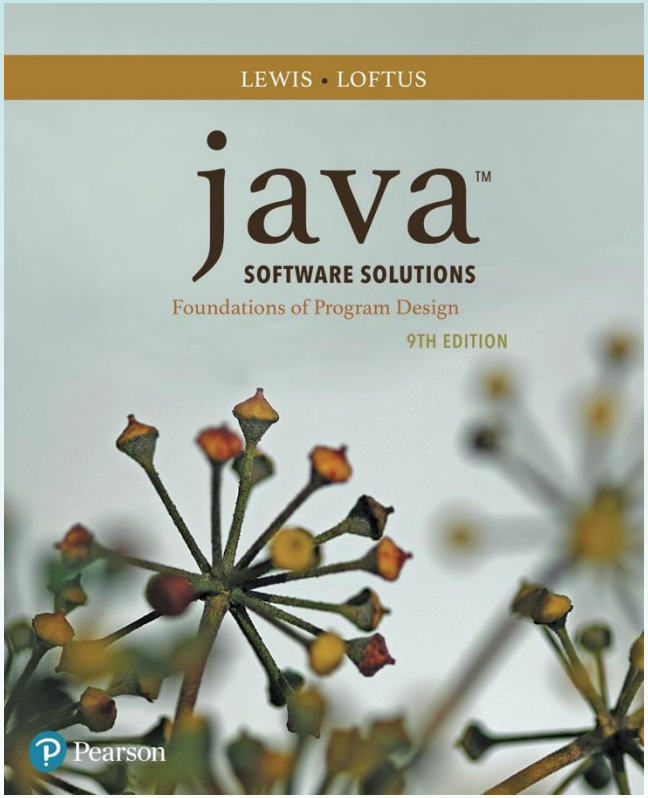
Radio Buttons

- To establish a set of mutually exclusive options, the radio buttons that work together as a group are added to a `ToggleGroup` object
- The `setToggleGroup` method is used to specify which toggle group a button belongs to
- The `isSelected` method of a radio button returns true if that button is currently "on"

Summary

- Chapter 5 focused on:
  - boolean expressions
  - the if and if-else statements
  - comparing data
  - while loops
  - iterators
  - the `ArrayList` class
  - more GUI controls

Chapter 6
More Conditionals and Loops



Java Software Solutions

Foundations of Program Design

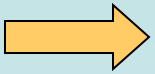
9th Edition

John Lewis
William Loftus

More Conditionals and Loops

- Now we can fill in some additional details regarding Java conditional and repetition statements
- Chapter 6 focuses on:
  - the `switch` statement
  - the conditional and `switch` expression
  - the `do` loop
  - the `for` loop
  - using conditionals and loops with graphics
  - graphic transformations

Outline



The `switch` Statement

The conditional and `switch` expression

The `do` Statement

The `for` Statement

Using Loops and Conditionals with Graphics

Graphic Transformations

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 list of values and a statement
- The flow of control transfers to statement associated with the first case value that matches
- There is also a switch expression

The switch Statement

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

switch and case are reserved words

```
switch ( expression )
{
    case value1 ->
        statement-1
    case value2 ->
        statement-2
    case value3 ->
        statement-3
    case  ...

}
```

If *expression* matches *value2*, control jumps to here

The switch Statement

- An example of a `switch` statement:

```
switch (option)
{
   case 'B'->
      aCount++;
   case 'C'->
      bCount++;
   case 'D'->
      cCount++;
   case 'a', 'e', 'i', 'o', 'u' ->
      vowels++;
}
```

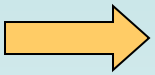
The switch Statement

- 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

- A `switch` expression type can be integers, characters, enumerated types, `String` and other objects
- You cannot use a `switch` with floating point or long values
- The implicit boolean condition in a `switch` statement is equality
- You cannot perform relational checks with a `switch` statement
- See `GradeReport.java`

Outline



The `switch` Statement

The conditional and `switch` expression

The `do` Statement

The `for` Statement

Using Loops and Conditionals with Graphics

Graphic Transformations

The Conditional Operator

- The *conditional operator* evaluates to one of two expressions based on a boolean condition
- 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

Quick Check

Express the following logic in a succinct manner using the conditional operator.

```
if (val <= 10)
   System.out.println("It is not greater than 10.");
else
   System.out.println("It is greater than 10.");
```

Quick Check

Express the following logic in a succinct manner using the conditional operator.

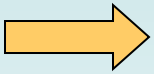
```
if (val <= 10)
   System.out.println("It is not greater than 10.");
else
   System.out.println("It is greater than 10.");


System.out.println("It is" +
   ((val <= 10) ? " not" : "") +
   " greater than 10.");
```

`switch` Expressions

- To choose among more than two values can use `switch` *expression*
- switch expression needs to be exhaustive or have a default clause

```
String seasonName = switch (seasonCode) {
    case 0 -> "Spring";
    case 1 -> "Summer";
    case 2 -> "Fall";
    case 3 -> "Winter";
    default -> "???";
};
```

Outline



The `switch` Statement

The conditional and `switch` expression

The `do` Statement

The `for` Statement

Using Loops and Conditionals with Graphics

Graphic Transformations

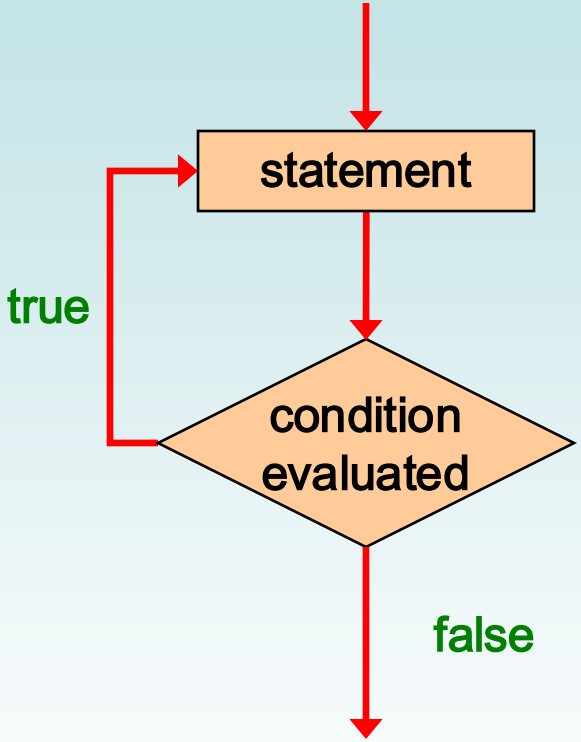
The do Statement

- A *do statement* has the following syntax:

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

- The **`statement-list`** 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

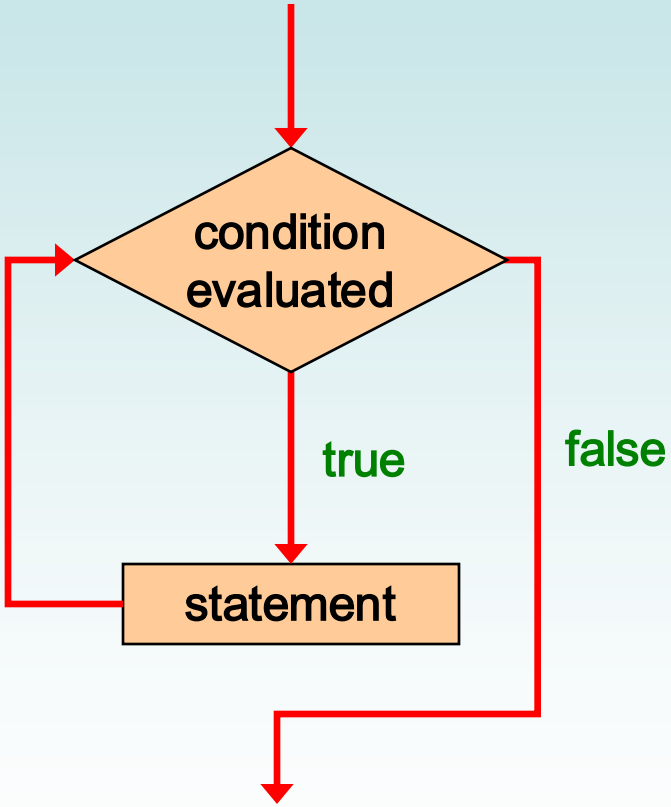
- An example of a `do` loop:

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

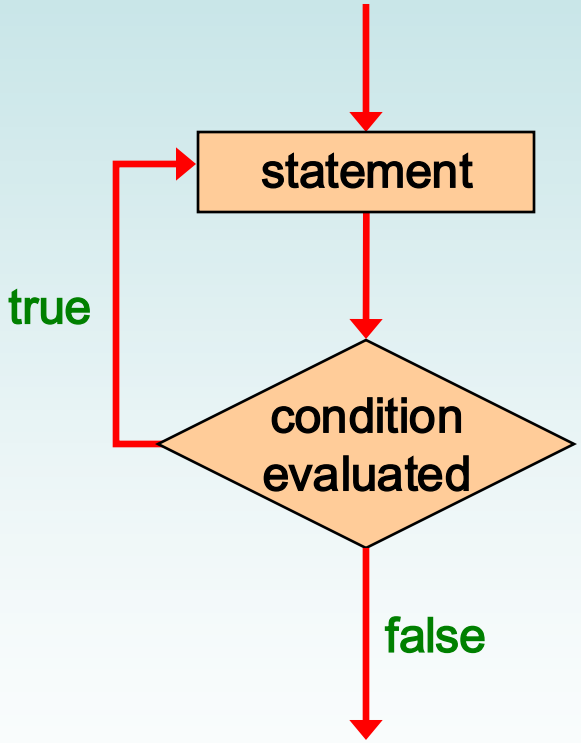
- The body of a `do` loop executes at least once
- See `ReverseNumber.java`

Comparing while and do

The while Loop



The do Loop

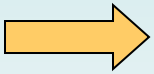


Outline

The `switch` Statement

The conditional and `switch` expression

The `do` Statement



The `for` Statement

Using Loops and Conditionals with Graphics

Graphic Transformations

The for Statement

- A *for statement* has the following syntax:

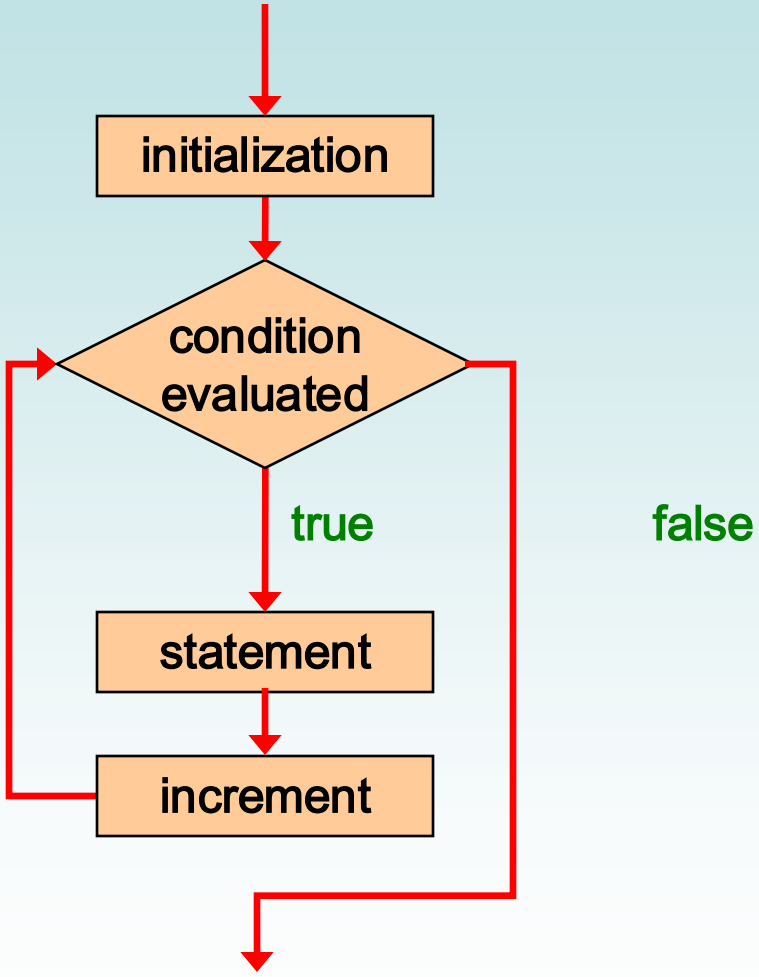
The ***initialization*** is executed once before the loop begins

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

Logic of a for loop



The for Statement

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

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

The for Statement

- 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

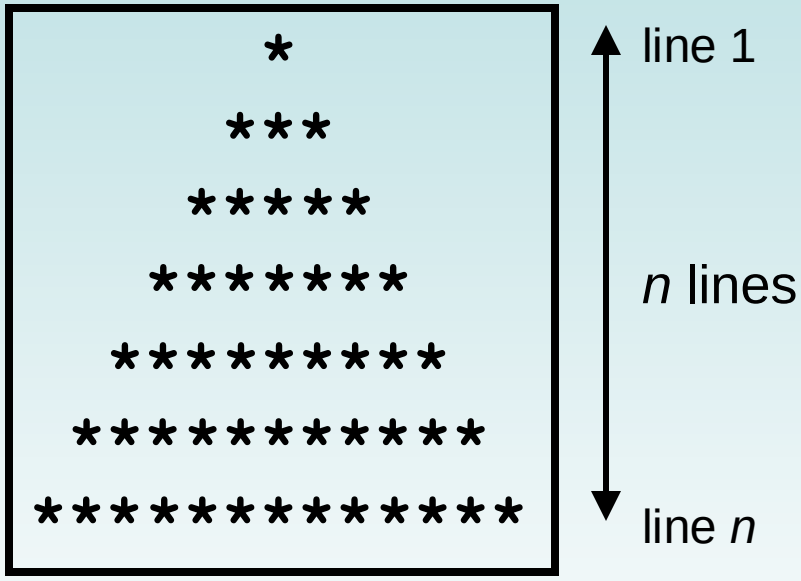
The for Statement

- 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
- See `Multiples.java`
- See `Stars.java`
- See also `Wedge.java` and `Tree.java`

Thought Process: Analyze Pattern



Program logic:
1. loop to print each line
2. inner loop print spaces
3. inner loop print stars

- Need to print ' ' and '*'
- How many ' ' for each line?
  - line *n* has 0 spaces
  - line *n*-1 has 1 space
  - (seems that line number plus number of spaces is *n*)
  - line 1 must have *n*-1 spaces
  - line *i* must have *n* - *i* spaces
- How many '*' for each line?
  - 1, 3, 5, … for lines 1, 2, 3, ---
  - line *i* seems to have 2**i*-1 '*'

Quick Check

Write a code fragment that rolls a die 100 times and counts the number of times a 3 comes up.

Quick Check

Write a code fragment that rolls a die 100 times and counts the number of times a 3 comes up.

```
Die die = new Die();
int count = 0;
for (int num=1; num <= 100; num++)
   if (die.roll() == 3)
      count++;
Sytem.out.println(count);
```

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

For-each Loops

- A variant of the `for` loop simplifies the repetitive processing of items in an iterator
- For example, suppose `bookList` is an `ArrayList<Book>` object
- The following loop will print each book:

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

- This version of a for loop is often called a *for-each loop*

For-each Loops

- A for-each loop can be used on any object that implements the `Iterable` interface
- It eliminates the need to retrieve an iterator and call the `hasNext` and `next` methods explicitly
- It also will be helpful when processing arrays, which are discussed in Chapter 8

Quick Check

Write a for-each loop that prints all of the `Student` objects in an `ArrayList<Student>` object called `roster`.

Quick Check

Write a for-each loop that prints all of the `Student` objects in an `ArrayList<Student>` object called `roster`.

```
for (Student student : roster)
   System.out.println(student);
```

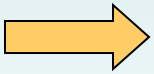
Outline

The `switch` Statement

The conditional and `switch` expression

The `do` Statement

The `for` Statement



Using Loops and Conditionals with Graphics

Graphic Transformations

More Graphics

- Conditionals and loops enhance our ability to generate interesting graphics
- See `Bullseye.java`
- See `Boxes.java`

Outline

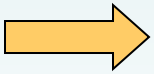
The `switch` Statement

The conditional and `switch` expression

The `do` Statement

The `for` Statement

Using Loops and Conditionals with Graphics



Graphic Transformations

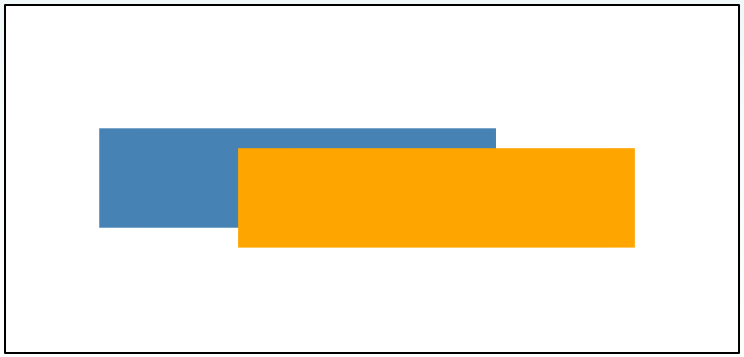
Graphic Transformations

- A JavaFX *transformation* changes the way a node is presented visually
  - *translation* – shifts the position along the x or y axis
  - *scaling* – causes the node to appear larger or smaller
  - *rotation* – rotates the node around its center point
  - *shearing* – rotates one axis so that the x and y axes are no longer perpendicular

Translation

- The following creates two rectangles in the same position, then shifts the second one:

```
Rectangle rec1 = new Rectangle(100, 100, 200, 50);
rec1.setFill(Color.STEELBLUE);

Rectangle rec2 = new Rectangle(100, 100, 200, 50);
rec2.setFill(Color.ORANGE);
rec2.setTranslateX(70);
rec2.setTranslateY(10);
```



Scaling

- The following displays two `ImageView` objects, the second scaled to 70%:

```
Image img = new Image("water lily.jpg");
ImageView imgView1 = new ImageView(img);

ImageView imgView2 = new ImageView(img);
imgView2.setX(300);
imgView2.setScaleX(0.7);
imgView2.setScaleY(0.7);
```

Rotation

- The parameter to `setRotate` determines how many degrees the node is rotated
- If the parameter positive, the node is rotated clockwise
- If the parameter is negative, the node is rotated counterclockwise

Rotation

```
Rectangle rec = new Rectangle(50, 100, 200, 50);
rec.setFill(Color.STEELBLUE);
rec.setRotate(40);

Text text = new Text(270, 125, "Tilted Text!");
text.setFont(new Font("Courier", 24));
text.setRotate(-15);
```

Rotation

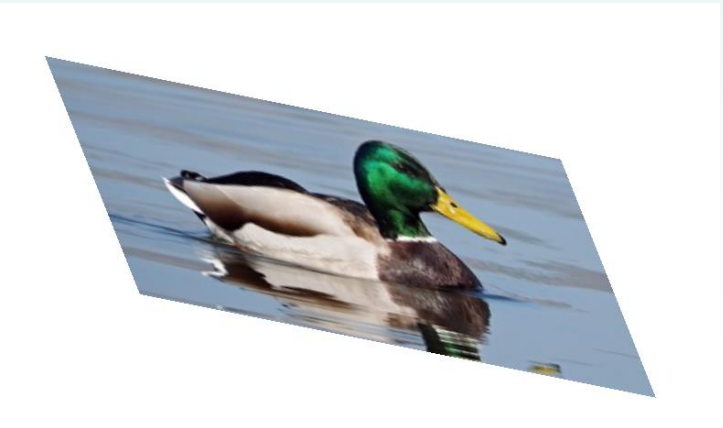
- To rotate a node around a point other than its center point, create a `Rotate` object and add it to the node's list of transformations
- The following rotates a node 45 degrees around the point (70, 150):

```
node.getTransforms().add(new Rotate(45, 70, 150));
```

Shearing

- Shearing is accomplished by creating a `Shear` object and adding it to this list of transformations
- The following applies a shear of 40% on the x axis and 20% on the y axis to an `ImageView` object:

```
Image img = new Image("duck.jpg");
ImageView imgView = new ImageView(img);
imgView.getTransforms().add(new Shear(0.4, 0.2));
```

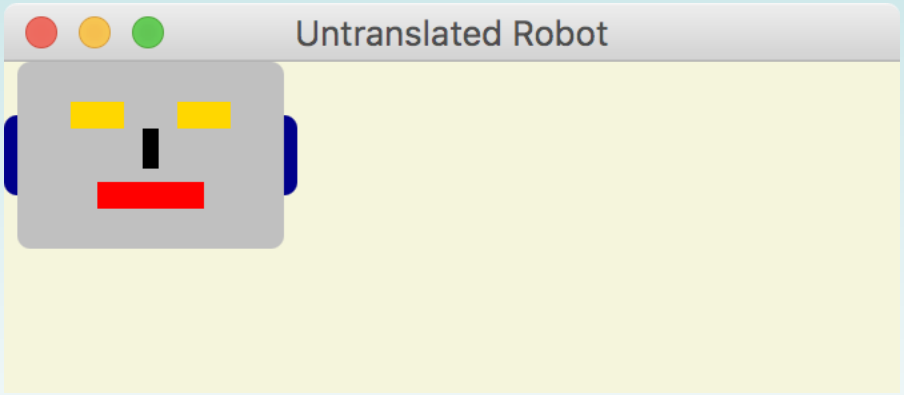


Transformations on Groups

- Transformations can be applied to any JavaFX nodes
    - shapes, images, controls
    - groups and panes
- When applied to a group or pane, the transformation is applied to each node it contains
- See `RobotFace.java`
- See `Robots.java`

Transformations on Groups

- If presented as defined, the robot face would be displayed in the upper left corner:



Summary

- Chapter 6 focused on:
  - the `switch` statement
  - the conditional and `switch` expression
  - the `do` loop
  - the `for` loop
  - using conditionals and loops with graphics
  - graphic transformations

1. Flow of Control:
   The order in which statements are executed in a program.
2. Boolean Expression:
   An expression that evaluates to either true or false.
3. if Statement:
   Executes a block of code if a condition is true.
   Example:
   ```
   if (age > 18) {
       System.out.println("Adult");
   }
   ```
4. if-else Statement:
   Chooses one of two blocks based on a Boolean condition.
   Example:
   ```
   if (score >= 50) {
       System.out.println("Pass");
   } else {
       System.out.println("Fail");
   }
   ```
5. Nested if Statement:
   An if statement placed inside another if or else clause to make further decisions.
6. Block Statement:
   A group of statements enclosed in braces { } to form a single compound statement.
7. Relational Operator (==):
   Tests if two values are equal.
8. Relational Operator (!=):
   Tests if two values are not equal.
9. Relational Operator (<):
   Tests if one value is less than another.
10. Relational Operator (>):
    Tests if one value is greater than another.
11. Relational Operator (<=):
    Tests if one value is less than or equal to another.
12. Relational Operator (>=):
    Tests if one value is greater than or equal to another.
13. Logical NOT Operator (!):
    Negates a boolean value. For example, !true yields false.
14. Logical AND Operator (&&):
    Returns true only if both operands are true.
15. Logical OR Operator (||):
    Returns true if at least one operand is true.
16. Short-circuit Evaluation:
    Logical operators (&&, ||) stop evaluating as soon as the result is determined.
17. Truth Table:
    A table listing all possible values of a Boolean expression based on every input combination.

18. if Statement Example:
    if (age > 18) { System.out.println("Adult"); }
19. if-else Statement Example:
    if (score >= 50) { System.out.println("Pass"); } else { System.out.println("Fail"); }
20. Ternary (Conditional) Operator:
    A shorthand for if-else that returns a value.
    Syntax: condition ? expression1 : expression2
21. Ternary Operator Example:
    int max = (a > b) ? a : b;
22. switch Statement:
    Selects a block of code to execute based on the value of an expression.
    Example:

```
switch(day) {
    case 1: System.out.println("Monday"); break;
    default: System.out.println("Other day");
}
```

23. switch Expression:
    A variant of the switch statement that returns a value (newer Java versions).
24. do-while Loop:
    A post-test loop that executes its body at least once before checking the condition.
    Syntax:

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

25. while Loop:
    A pre-test loop that executes as long as the condition is true.
    Syntax:

```
while (condition) {
    statements;
}
```

26. for Loop:
    A loop that includes initialization, condition, and increment/decrement in one statement.
    Syntax:

```
for (initialization; condition; increment) {
    statements;
}
```

27. for-each Loop:
    A simplified loop for iterating over arrays or collections.
    Syntax:

```
for (Type item : collection) {
    statements;
}
```

28. Infinite Loop:
    A loop that never terminates because its condition always evaluates to true.

29. Sentinel Value:
    A special value used to indicate the end of input within a loop.
30. Input Validation Loop:
    A loop that continues to prompt the user until valid input is received.
    Example:
    ```
    do {
        System.out.print("Enter a positive number: ");
        input = scanner.nextInt();
    } while (input <= 0);
    ```
31. Nested Loops:
    Loops inside loops, useful for multi-dimensional data processing.
    Example (nested while):
    ```
    int i = 1;
    while (i <= 3) {
        int j = 1;
        while (j <= 2) {
            System.out.println(i + "," + j);
            j++;
        }
        i++;
    }
    ```
32. Comparing Floating Point Numbers:
    Due to precision issues, use a tolerance when comparing floats.
33. Tolerance in Floating Point Comparison:
    A small value (e.g., 0.00001) used to determine if two floats are “close enough.”
    Example:
    ```
    if (Math.abs(f1 - f2) < 0.00001) { … }
    ```
34. Comparing Characters:
    Uses relational operators; characters are compared based on their Unicode values.
35. Unicode Ordering:
    The natural order of characters in Java: digits come first, then uppercase letters, then lowercase letters.
36. Comparing Strings using equals():
    Method to check if two strings are identical in content.
    Example:
    ```
    if (str1.equals(str2)) { … }
    ```
37. Comparing Strings using compareTo():
    Method to determine lexicographic order between strings.
    Example:
    ```
    int cmp = str1.compareTo(str2);
    if (cmp < 0) { … }
    ```
38. Lexicographic Ordering:
    Ordering based on dictionary sequence, which is affected by case and length.

39. Comparing Objects:
    By default, equals() compares object references, but it can be overridden to compare object content.
40. Assignment Operator (=) vs Equality Operator (==):
    '=' assigns a value, while '==' checks if two values are equal.
41. Indentation Importance:
    Proper indentation is essential for code readability, though it does not affect how Java executes the code.
42. Boolean Literal: true
    The literal representing a true value.
43. Boolean Literal: false
    The literal representing a false value.
44. Operator Precedence:
    Defines the order in which operators are evaluated (arithmetic > relational > logical).
45. Compound Conditions:
    Combining multiple expressions using logical operators (&&, ||, !).
46. Code Sample Using &&:
    ```
    if (x > 0 && y > 0) { System.out.println("Both positive"); }
    ```
47. Code Sample Using ||:
    ```
    if (x < 0 || y < 0) { System.out.println("At least one negative"); }
    ```
48. Code Sample: Nested Conditions:
    ```
    if (a > b) {
        if (a > c) { System.out.println("a is largest"); }
    }
    ```
49. Code Sample: while Loop Counting:
    ```
    int count = 1;
    while (count <= 10) {
        System.out.println(count);
        count++;
    }
    ```
50. Code Sample: do-while Loop Counting:
    ```
    int count = 1;
    do {
        System.out.println(count);
        count++;
    } while (count <= 10);
    ```
51. Code Sample: for Loop Iteration:
    ```
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
    ```
52. Code Sample: for-each Loop Iteration:
    ```
    for (String item : items) {
        System.out.println(item);
    }
    ```

53. Code Sample: Ternary Operator Usage:
    String result = (score >= 60) ? "Pass" : "Fail";
54. Code Sample: switch Statement:
    switch(day) {
        case 1: System.out.println("Monday"); break;
        case 2: System.out.println("Tuesday"); break;
        default: System.out.println("Other day");
    }
55. Flowchart for if Statement:
    A conceptual diagram that shows the decision process in an if statement.
56. Decision Making in Java:
    Using conditionals (if, if-else, switch) to control program flow.
57. Conditional Statements Overview:
    Statements that let you choose which code to execute based on conditions.
58. Repetition Statements Overview:
    Loops that allow code to be executed repeatedly.
59. Loop Control Statements:
    Keywords like break and continue (not detailed here) that alter loop behavior.
60. Pre-test Loop:
    A loop (while, for) that tests its condition before executing the loop body.
61. Post-test Loop:
    A loop (do-while) that tests its condition after executing the loop body.
62. Increment Operator (++):
    Increases a variable's value by one.
    Example: a++;
63. Decrement Operator (--):
    Decreases a variable's value by one.
    Example: a--;
64. Code Sample: Increment Operator:
    int a = 5;
    a++; // a becomes 6
65. Code Sample: Decrement Operator:
    int a = 5;
    a--; // a becomes 4
66. Infinite Loop Problem:
    Occurs when a loop's condition is never false, causing non-termination.
67. Debugging Loop Termination:
    The process of ensuring that loop conditions will eventually evaluate to false.
68. Calculating Loop Iterations:
    Determining how many times a loop will execute, especially in nested loops.
69. Code Sample: Nested while Loops:
    int i = 1;
    while (i <= 3) {
        int j = 1;
        while (j <= 2) {

```
                System.out.println(i + "," + j);
                j++;
            }
            i++;
        }
```

70. Code Sample: Nested for Loops:

```
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 2; j++) {
                System.out.println(i + "," + j);
            }
        }
```

71. Using Logical Operators in Conditions:

    Combining multiple conditions with &&, ||, and !.

72. Code Sample: if with && and ||:

```
        if ((x > 0 && y > 0) || z == 0) {
            System.out.println("Condition met");
        }
```

73. Code Sample: Input Validation using while:

```
        Scanner sc = new Scanner(System.in);
        int input;
        do {
            System.out.print("Enter a positive number: ");
            input = sc.nextInt();
        } while (input <= 0);
```

74. Short-Circuit Behavior Example:

    In the expression if (false && someMethod()), someMethod() is never called.

75. Code Sample: Avoiding Division by Zero:

```
        if (count != 0 && total / count > MAX) {
            System.out.println("Safe division");
        }
```

76. Comparing Data and Data Types:

    Understanding that comparisons differ based on data types (int, float, char, String, etc.).

77. Code Sample: Comparing Two Numbers:

```
        if (a < b) {
            System.out.println("a is less than b");
        }
```

78. Code Sample: Comparing Characters:

```
        if (ch1 < ch2) {
            System.out.println(ch1 + " comes before " + ch2);
        }
```

79. Code Sample: Using equals() for Strings:

```
        if (str1.equals(str2)) {
            System.out.println("Strings are equal");
        }
```

80. Code Sample: Using compareTo() for Strings:

```
int cmp = str1.compareTo(str2);
if (cmp < 0) {
    System.out.println(str1 + " comes before " + str2);
}
```

81. Importance of Tolerance for Floats:
    Always use a tolerance value when comparing floating-point numbers.
82. Code Sample: Floating Point Comparison:

```
if (Math.abs(f1 - f2) < 0.00001) {
    System.out.println("Floats are essentially equal");
}
```

83. for-each Loop in Arrays:
    Using for-each to iterate over array elements.
    Example:

```
for (int num : numbers) {
    System.out.println(num);
}
```

84. Iterator Interface Overview:
    An object that allows sequential access to elements in a collection.
85. Difference Between Pre-test and Post-test Loops:
    Pre-test loops check the condition before executing the loop; post-test loops execute the body first.
86. Use of Parentheses in Complex Boolean Expressions:
    Parentheses help clarify the order of evaluation in compound conditions.
87. Code Sample: if with Parentheses:

```
if ((a > b) && (c < d)) {
    System.out.println("Condition met");
}
```

88. Use of Curly Braces for Block Statements:
    Always use { } to group multiple statements, especially in conditionals and loops.
89. Code Sample: Block Statement Example:

```
if (x > y) {
    System.out.println("x is greater");
    x--;
} else {
    System.out.println("y is greater or equal");
}
```

90. Nested if-else Complexities:
    Understanding how else clauses pair with the nearest unmatched if.
91. Matching else Clause to Nearest if:
    A rule in Java where an else is associated with the closest preceding if that has not been paired with an else.
92. Common Pitfalls in Nested if Statements:
    Errors such as misaligned braces that lead to unexpected behavior.

93. Code Sample: Nested if Statement (MinOfThree):

```
if (num1 < num2) {
    if (num1 < num3) {
        min = num1;
    } else {
        min = num3;
    }
} else {
    if (num2 < num3) {
        min = num2;
    } else {
        min = num3;
    }
}
```

94. Self-Review Question: What is Flow of Control?

    It is the sequence in which statements are executed in a program.

95. Self-Review Question: What is a Truth Table?

    A chart that displays the output of a Boolean expression for every possible input combination.

96. Self-Review Question: How Do Logical Operators Work?

    They combine Boolean values: && returns true if both are true, || returns true if at least one is true, and ! negates a Boolean.

97. Self-Review Question: Difference Between while and do-while Loops?

    A while loop tests its condition before executing (and might not run at all), whereas a do-while loop executes its body at least once before checking the condition.

98. Self-Review Question: Write a Code Fragment to Count Occurrences of a Value.

    Example:

```
int count = 0;
for (int i = 0; i < array.length; i++) {
    if (array[i] == target) {
        count++;
    }
}
System.out.println(count);
```

99. Use of break Statement (Loop Control):

    Terminates the loop immediately when executed.

100. Use of continue Statement (Loop Control):

    Skips the remaining statements in the loop body and continues with the next iteration.

101. Code Sample: Using break in a Loop:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    System.out.println(i);
}
```

102. Code Sample: Using continue in a Loop:

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) continue;
```

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

103. Repetition Statement Overview:
    A loop allows a program to execute a block of code repeatedly based on a condition.
104. Code Sample: Counting Using a for Loop with Custom Increment:

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

105. Importance of Testing Loop Conditions:
    Ensure that the loop’s condition will eventually become false to avoid infinite loops.
106.