References

Decision Structures and Loop

References:

Tony Gaddis, Chapters 3 and 4, Starting out with Java: From Control Structures through Objects, 7 edition

Chapter Topics (1 of 2)

- The `if` Statement
- The `if-else` Statement
- Nested `if` statements
- The `if-else-if` Statement
- Logical Operators
- More about Variable Declaration and Scope
- The Conditional Operator
- The `switch` Statement
- Displaying Formatted Output with `System.out.printf`

Chapter Topics (2 of 2)

- The Increment and Decrement Operators
- The `while` Loop
- Using the `while` Loop for Input Validation
- The `do-while` Loop
- The `for` Loop
- Running Totals and Sentinel Values
- Nested Loops
- The `break` and `continue` Statements
- Deciding Which Loop to Use

3

The `if` Statement

- The `if` statement decides whether a section of code executes or not.
- The `if` statement uses a `boolean` to decide whether the next statement or block of statements executes.

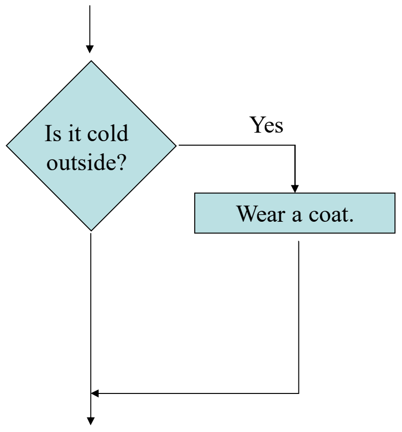
if (boolean expression is true)
  execute next statement.

4

Flowcharts (1 of 2)

- If statements can be modeled as a flow chart.

```
if (coldOutside)
   wearCoat();
```



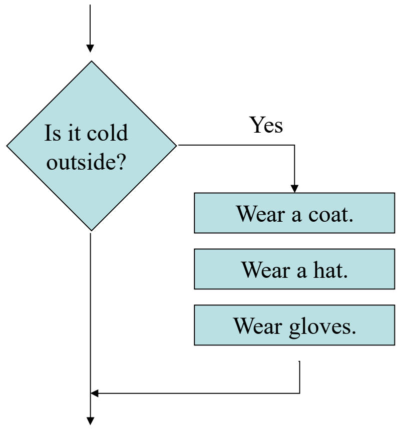
5

Flowcharts (2 of 2)

- A block `if` statement may be modeled as:

```
if (coldOutside)
{
   wearCoat();
   wearHat();
   wearGloves();
}
```

Note the use of curly braces to block several statements together.



6

Relational Operators

- In most cases, the `boolean` expression, used by the `if` statement, uses *relational operators.*

Relational Operator	Meaning
`>`	is greater than
`<`	is less than
`>=`	is greater than or equal to
`<=`	is less than or equal to
`==`	is equal to
`!=`	is not equal to

7

Boolean Expressions

- A *boolean expression* is any variable or calculation that results in a *true* or *false* condition.

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y.
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

8

`if` Statements and Boolean Expressions

```
if (x > y)
    System.out.println("X is greater than Y");

if(x == y)
    System.out.println("X is equal to Y");

if(x != y)
{
    System.out.println("X is not equal to Y");
    x = y;
    System.out.println("However, now it is.");
}
```

Example: AverageScore.java

9

Programming Style and `if` Statements (1 of 2)

- An `if` statement can span more than one line; however, it is still one statement.

```
if (average > 95)
    grade = 'A';
```

is functionally equivalent to

```
if(average > 95) grade = 'A';
```

10

Programming Style and `if` Statements (2 of 2)

- Rules of thumb:
  - The conditionally executed statement should be on the line after the `if` condition.
  - The conditionally executed statement should be indented one level from the `if` condition.
  - If an `if` statement does not have the block curly braces, it is ended by the first semicolon encountered after the `if` condition.

```
if (expression)        ← No semicolon here.
   statement;          ← Semicolon ends statement here.
```

Pearson

11

Block `if` Statements (1 of 2)

- Conditionally executed statements can be grouped into a block by using curly braces { } to enclose them.
- If curly braces are used to group conditionally executed statements, the `if` statement is ended by the closing curly brace.

```
if (expression)
{
  statement1;
  statement2;
}       ← Curly brace ends the statement.
```

Pearson

12

Block `if` Statements (2 of 2)

- Remember that when the curly braces are not used, then only the next statement after the `if` condition will be executed conditionally.

```
if (expression)
   statement1;   ← Only this statement is conditionally executed.
   statement2;
   statement3;
```

13

Flags

- A flag is a `boolean` variable that monitors some condition in a program.
- When a condition is true, the flag is set to `true`.
- The flag can be tested to see if the condition has changed.

```
if (average > 95)
highScore = true;
```

- Later, this condition can be tested:

```
if (highScore)
System.out.println("That's a high score!");
```

14

Comparing Characters

- Characters can be tested with relational operators.
- Characters are stored in memory using the Unicode character format.
- Unicode is stored as a sixteen (16) bit number.
- Characters are *ordinal*, meaning they have an order in the Unicode character set.
- Since characters are ordinal, they can be compared to each other.

```
char c = 'A';
      if(c < 'Z')
      System.out.println("A is less than Z");
```

15

`if-else` Statements

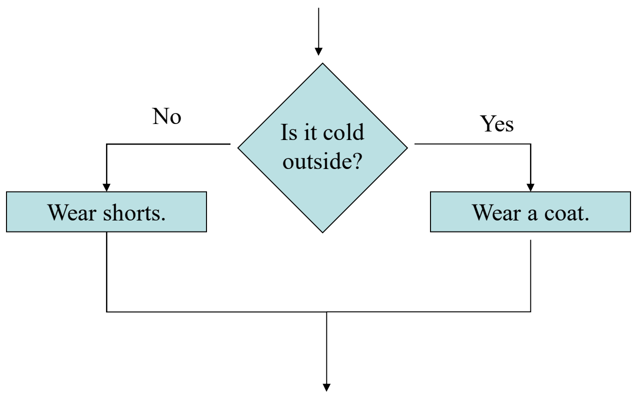
- The `if-else` statement adds the ability to conditionally execute code when the `if` condition is false.

```
if (expression)
statementOrBlockIfTrue;
else
statementOrBlockIfFalse;
```

- See example: Division.java

16

`if-else` Statement Flowcharts



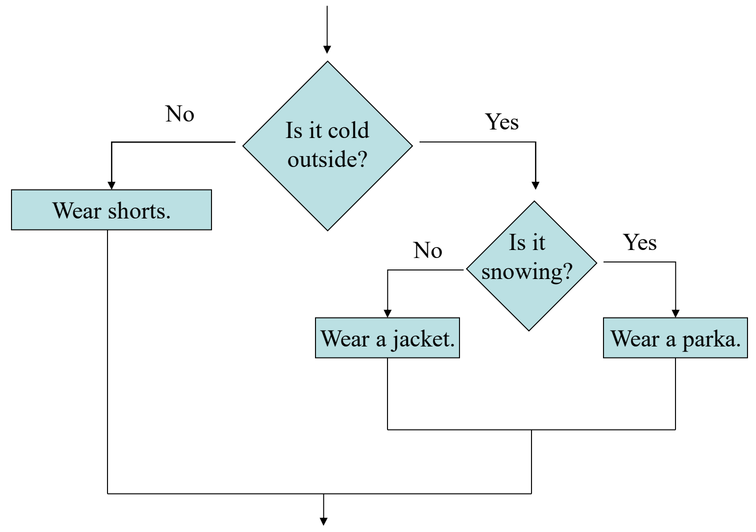
17

Nested `if` Statements

- If an `if` statement appears inside another `if` statement (single or block) it is called a *nested* `if` statement.
- The nested `if` is executed only if the outer `if` statement results in a true condition.
- See example: LoanQualifier.java

18

Nested `if` Statement Flowcharts



19

Nested `if` Statements

```
if (coldOutside)
{
     if (snowing)
     {
          wearParka();
     }
     else
     {
          wearJacket();
     }
}
else
{
     wearShorts();
}
```

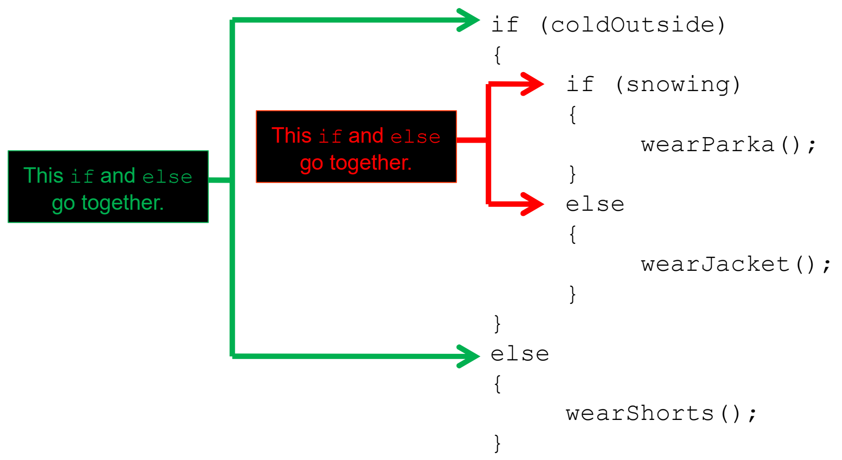
20

`if-else` Matching

- Curly brace use is not required if there is only one statement to be conditionally executed.
- However, sometimes curly braces can help make the program more readable.
- Additionally, proper indentation makes it much easier to match up else statements with their corresponding `if` statement.

21

Alignment and Nested `if` Statements



```
if (coldOutside)
{
    if (snowing)
    {
        wearParka();
    }
    else
    {
        wearJacket();
    }
}
else
{
    wearShorts();
}
```

22

if-else-if Statements (1 of 2)

```
if (expression_1)
{
   statement;  }  If expression_1 is true these statements are
   statement;  }  executed, and the rest of the structure is ignored.
   etc.
}
else if (expression_2)
{
   statement;  }  Otherwise, if expression_2 is true these statements are
   statement;  }  executed, and the rest of the structure is ignored.
   etc.
}
```

Insert as many else if clauses as necessary

```
else
{
   statement;  }  These statements are executed if none of the
   statement;  }  expressions above are true.
   etc.
}
```

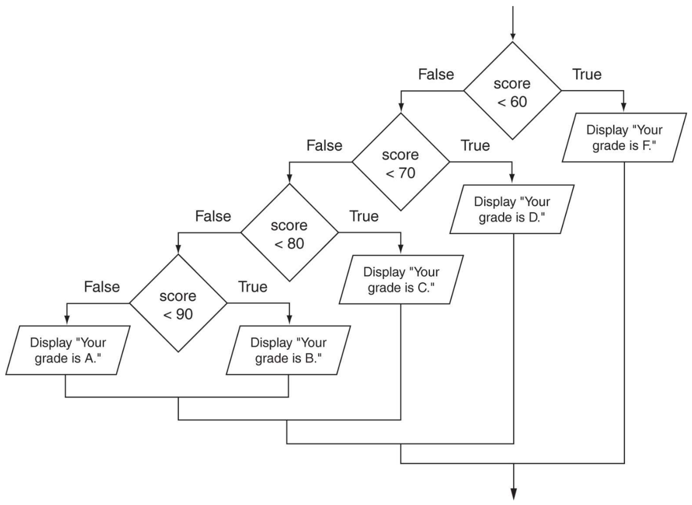
23

if-else-if Statements (2 of 2)

- Nested `if` statements can become very complex.
- The `if-else-if` statement makes certain types of nested decision logic simpler to write.
- Care must be used since `else` statements match up with the immediately preceding unmatched `if` statement.
- See example: TestResults.java

24

`if-else-if` Flowchart


score < 60
True
False
Display "Your grade is F."
score < 70
Display "Your grade is D."
score < 80
Display "Your grade is C."
score < 90
Display "Your grade is B."
Display "Your grade is A."

25

Logical Operators (1 of 2)

- Java provides two binary *logical operators* (`&&` and `||`) that are used to combine `boolean` expressions.
- Java also provides one *unary* (`!`) logical operator to reverse the truth of a `boolean` expression.

26

Logical Operators (2 of 2)

Operator	Meaning	Effect
`&&`	AND	Connects two `boolean` expressions into one. Both expressions must be true for the overall expression to be true.
`||`	OR	Connects two `boolean` expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which one.
`!`	NOT	The ! operator reverses the truth of a `boolean` expression. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

27

The `&&` Operator

- The logical AND operator (`&&`) takes two operands that must both be `boolean` expressions.
- The resulting combined expression is true if (and *only* if) both operands are true.
- See example: LogicalAnd.java

Expression 1	Expression 2	Expression1 && Expression2
true	false	false
false	true	false
false	false	false
true	true	true

28

The || Operator

- The logical OR operator (`||`) takes two operands that must both be `boolean` expressions.
- The resulting combined expression is false if (and *only* if) both operands are false.
- Example: LogicalOr.java

Expression 1	Expression 2	Expression1 || Expression2
true	false	true
false	true	true
false	false	false
true	true	true

29

The ! Operator

- The `!` operator performs a logical NOT operation.
- If an *`expression`* is true, `!`*`expression`* will be false.

```
   if (!(temperature > 100))
         System.out.println("Below the maximum
temperature.");
```

- If **`temperature > 100`** evaluates to false, then the output statement will be run.

Expression 1	!Expression1
true	false
false	true

30

Short Circuiting

- Logical AND and logical OR operations perform *short-circuit evaluation* of expressions.
- Logical AND will evaluate to false as soon as it sees that one of its operands is a false expression.
- Logical OR will evaluate to true as soon as it sees that one of its operands is a true expression.

31

Order of Precedence (1 of 2)

- The `!` operator has a higher order of precedence than the `&&` and `||` operators.
- The `&&` and `||` operators have a lower precedence than relational operators like `<` and `>`.
- Parenthesis can be used to force the precedence to be changed.

32

Order of Precedence (2 of 2)

Order of Precedence	Operators	Description
1	**(unary negation) !**	Unary negation, logical NOT
2	`* / %`	Multiplication, Division, Modulus
3	`+ -`	Addition, Subtraction
4	`< > <= >=`	Less-than, Greater-than, Less-than or equal to, Greater-than or equal to
5	`== !=`	Is equal to, Is not equal to
6	`&&`	Logical AND
7	`||`	Logical NOT
8	`= += -=` `*= /= %=`	Assignment and combined assignment operators.

Variable Scope

- In Java, a local variable does not have to be declared at the beginning of the method.
- The scope of a local variable begins at the point it is declared and terminates at the end of the method.
- When a program enters a section of code where a variable has scope, that variable has *come into scope*, which means the variable is visible to the program.
- See example: VariableScope.java

The Conditional Operator (1 of 4)

- The *conditional operator* is a ternary (three operand) operator.
- You can use the conditional operator to write a simple statement that works like an `if-else` statement.

Pearson

35

The Conditional Operator (2 of 4)

- The format of the operators is:

```
BooleanExpression ? Value1 : Value2
```

- This forms a conditional expression.
- If *`BooleanExpression`* is true, the value of the conditional expression is *`Value1`*.
- If *`BooleanExpression`* is false, the value of the conditional expression is *`Value2`*.

Pearson

36

The Conditional Operator (3 of 4)

- Example:

```
z = x > y ? 10 : 5;
```

- This line is functionally equivalent to:

```
if(x > y)
  z = 10;
else
  z = 5;
```

37

The Conditional Operator (4 of 4)

- Many times the conditional operator is used to supply a value.

```
number = x > y ? 10 : 5;
```

- This is functionally equivalent to:

```
if(x > y)
number = 10;
else
number = 5;
```

- See example: ConsultantCharges.java

38

The `switch` Statement (1 of 4)

- The `if-else` statement allows you to make true / false branches.
- The `switch` statement allows you to use an ordinal value to determine how a program will branch.
- The `switch` statement can evaluate an *integer* type or *character* type variable and make decisions based on the value.

39

The `switch` Statement (2 of 4)

- The `switch` statement takes the form:

```
switch (SwitchExpression)
{
  case CaseExpression:
    // place one or more statements here
    break;
  case CaseExpression:
    // place one or more statements here
    break;

    // case statements may be repeated
    //as many times as necessary
  default:
    // place one or more statements here
}
```

40

The `switch` Statement (3 of 4)

```
switch (SwitchExpression)
{
  …
}
```

- The `switch` statement will evaluate the *SwitchExpression*, which can be a `byte`, `short`, `int`, `long`, or `char`. If you are using Java 7, the *SwitchExpression* can also be a string.
- If there is an associated `case` statement that matches that value, program execution will be transferred to that `case` statement.

41

The `switch` Statement (4 of 4)

- Each `case` statement will have a corresponding *CaseExpression* that must be unique.

```
case CaseExpression:
  // place one or more statements here
  break;
```

- If the *SwitchExpression* matches the *CaseExpression*, the Java statements between the colon and the `break` statement will be executed.

42

The case Statement

- The `break` statement ends the `case` statement.
- The `break` statement is optional.
- If a `case` does not contain a `break`, then program execution continues into the next `case`.
  - See example: NoBreaks.java
  - See example: PetFood.java
- The `default` section is optional and will be executed if no *CaseExpression* matches the *SwitchExpression*.
- See example: SwitchDemo.java

43

The System.out.printf Method (1 of 15)

- You can use the `System.out.printf` method to perform formatted console output.
- The general format of the method is:

```
System.out.printf(FormatString,
  ArgList);
```

44

The `System.out.printf` Method (2 of 15)

```
System.out.printf(FormatString, ArgList);
```

FormatString **is a string that contains text and/or special formatting specifiers.**

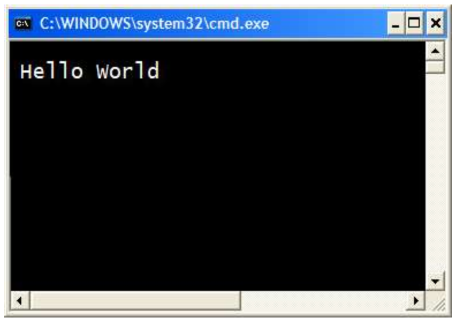
ArgList **is optional. It is a list of additional arguments that will be formatted according to the format specifiers listed in the format string.**

45

The `System.out.printf` Method (3 of 15)

- A simple example:

```
System.out.printf("Hello World\n");
```



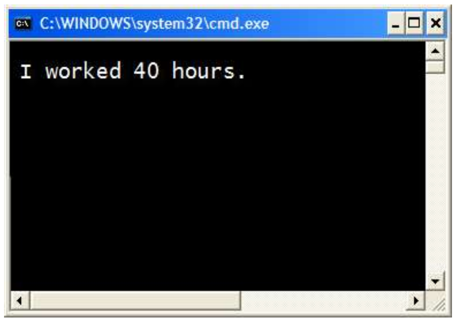
```
Hello World
```

46

The `System.out.printf` Method (4 of 15)

- Another example:

```
int hours = 40;
System.out.printf("I worked %d hours.\n", hours);
```

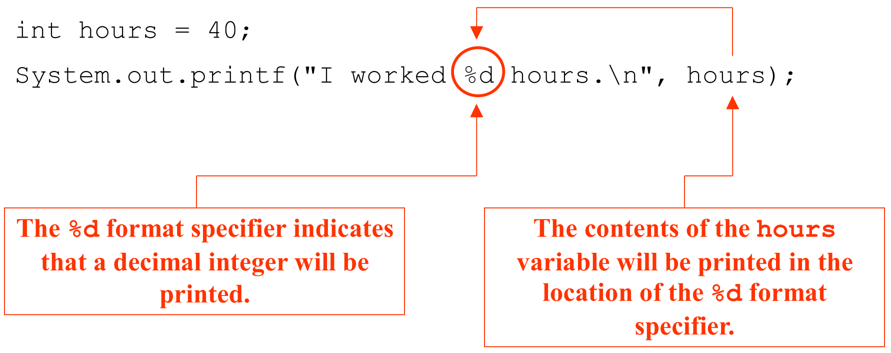

C:\WINDOWS\system32\cmd.exe

```
I worked 40 hours.
```

Pearson

47

The `System.out.printf` Method (5 of 15)



```
int hours = 40;
System.out.printf("I worked %d hours.\n", hours);
```

The %d format specifier indicates that a decimal integer will be printed.
The contents of the hours variable will be printed in the location of the %d format specifier.

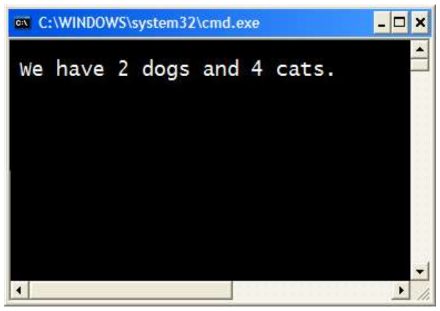
Pearson

48

The `System.out.printf` Method (6 of 15)

- Another example:

```
int dogs = 2, cats = 4;
System.out.printf("We have %d dogs and %d
                   cats.\n", dogs, cats);
```



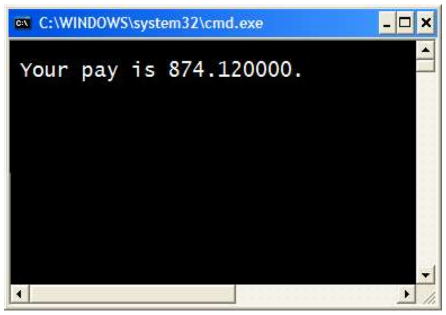
```
We have 2 dogs and 4 cats.
```

49

The `System.out.printf` Method (7 of 15)

- Another example:

```
double grossPay = 874.12;
System.out.printf("Your pay is %f.\n", grossPay);
```



```
Your pay is 874.120000.
```

50

The `System.out.printf` Method (8 of 15)

- Another example:

```
double grossPay = 874.12;
System.out.printf("Your pay is %f.\n", grossPay);
```

The `%f` format specifier indicates that a floating-point value will be printed.

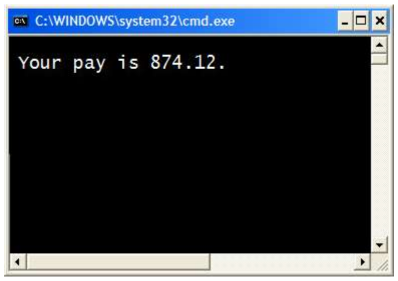
The contents of the `grossPay` variable will be printed in the location of the `%f` format specifier.

51

The `System.out.printf` Method (9 of 15)

- Another example:

```
double grossPay = 874.12;
System.out.printf("Your pay is %.2f.\n",
                  grossPay);
```



```
Your pay is 874.12.
```

52

The `System.out.printf` Method (10 of 15)

- Another example:

```
double grossPay = 874.12;
System.out.printf("Your pay is %.2f\n", grossPay);
```

The `%.2f` format specifier indicates that a floating-point value will be printed, rounded to two decimal places.

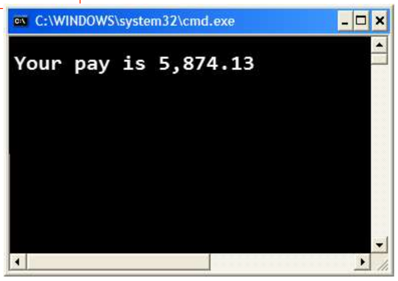
53

The `System.out.printf` Method (11 of 15)

- Another example:

```
double grossPay = 5874.127;
System.out.printf("Your pay is %,.2f\n", grossPay);
```

The `%,.2f` format specifier indicates that a floating-point value will be printed with comma separators, rounded to two decimal places.



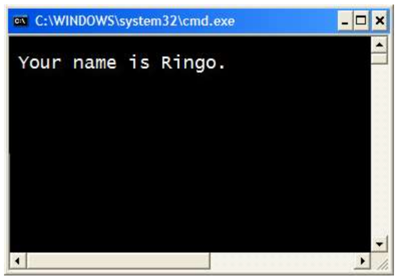
```
Your pay is 5,874.13
```

54

The System.out.printf Method (12 of 15)

- Another example:

```
String name = "Ringo";
System.out.printf("Your name is %s.\n", name);
```


C:\WINDOWS\system32\cmd.exe

```
Your name is Ringo.
```

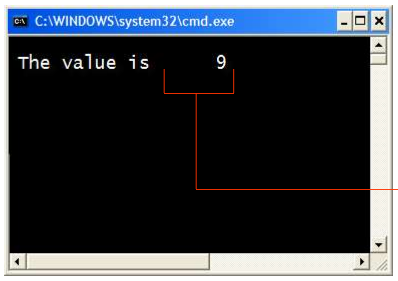
The %s format specifier indicates that a string will be printed.

55

The System.out.printf Method (13 of 15)

- Specifying a field width:

```
int number = 9;
System.out.printf("The value is %6d\n", number);
```


C:\WINDOWS\system32\cmd.exe

```
The value is      9
```

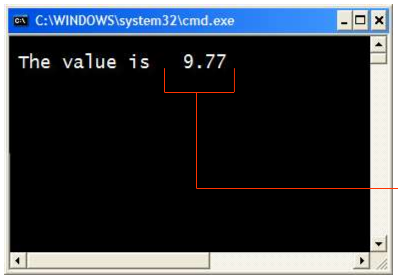
The %6d format specifier indicates the integer will appear in a field that is 6 spaces wide.

56

The `System.out.printf` Method (14 of 15)

- Another example:

```
double number = 9.76891;
System.out.printf("The value is %6.2f\n", number);
```



```
The value is   9.77
```

The `%6.2f` format specifier indicates the number will appear in a field that is 6 spaces wide, and be rounded to 2 decimal places.

The `System.out.printf` Method (15 of 15)

- See examples:
  - Columns.java
  - CurrencyFormat.java

The Increment and Decrement Operators

- There are numerous times where a variable must simply be incremented or decremented.

```
number = number + 1;
number = number – 1;
```

- Java provide shortened ways to increment and decrement a variable’s value.
- Using the **++** or **--** unary operators, this task can be completed quickly.

```
number++;  or  ++number;
number--;  or  --number;
```

- Example: IncrementDecrement.java

59

Differences Between Prefix and Postfix

- When an increment or decrement are the only operations in a statement, there is no difference between prefix and postfix notation.
- When used in an expression:
  - prefix notation indicates that the variable will be incremented or decremented prior to the rest of the equation being evaluated.
  - postfix notation indicates that the variable will be incremented or decremented after the rest of the equation has been evaluated.
- Example: Prefix.java

60

The `while` Loop (1 of 2)

- Java provides three different looping structures.
- The `while` loop has the form:

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

- While the condition is true, the statements will execute repeatedly.
- The `while` loop is a *pretest* loop, which means that it will test the value of the condition prior to executing the loop.

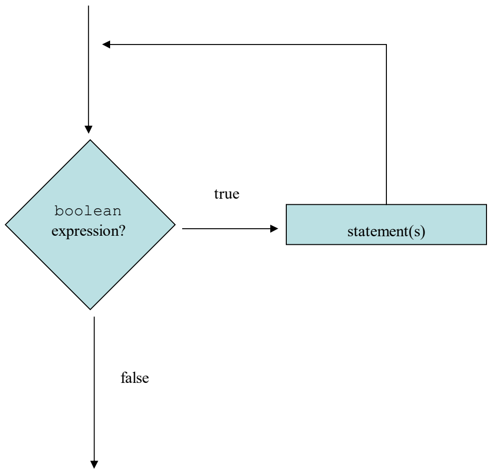
61

The `while` Loop (2 of 2)

- Care must be taken to set the condition to false somewhere in the loop so the loop will end.
- Loops that do not end are called *infinite loops.*
- A `while` loop executes 0 or more times. If the condition is false, the loop will not execute.
- Example: WhileLoop.java

62

The **while** loop Flowchart



63

Infinite Loops (1 of 2)

- In order for a `while` loop to end, the condition must become false. The following loop will not end:

```
int x = 20;
while(x > 0)
{
  System.out.println("x is greater than
                     0");
}
```

- The variable `x` never gets decremented so it will always be greater than 0.
- Adding the **`x--`** above fixes the problem.

64

Infinite Loops (2 of 2)

- This version of the loop decrements x during each iteration:

```
int x = 20;
while(x > 0)
{
   System.out.println("x is greater
                        than 0");
   x--;
}
```

65

Block Statements in Loops

- Curly braces are required to enclose block statement while loops. (like block if statements)

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

66

The `while` Loop for Input Validation

- *Input validation* is the process of ensuring that user input is valid.

```
System.out.print("Enter a number in the " +
                 "range of 1 through 100: ");
number = keyboard.nextInt();
// Validate the input.
while (number < 1 || number > 100)
{
  System.out.println("That number is invalid.");
  System.out.print("Enter a number in the " +
                   "range of 1 through 100: ");
  number = keyboard.nextInt();
}
```

- Example: SoccerTeams.java

67

The `do-while` Loop

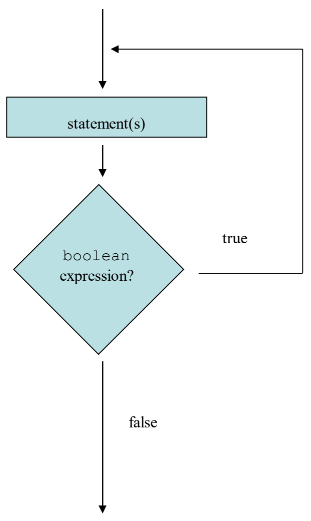
- The `do-while` loop is a *post-test* loop, which means it will execute the loop prior to testing the condition.
- The `do-while` loop (sometimes called a `do` loop) takes the form:

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

- Example: TestAverage1.java

68

The do-while Loop Flowchart



69

The for Loop

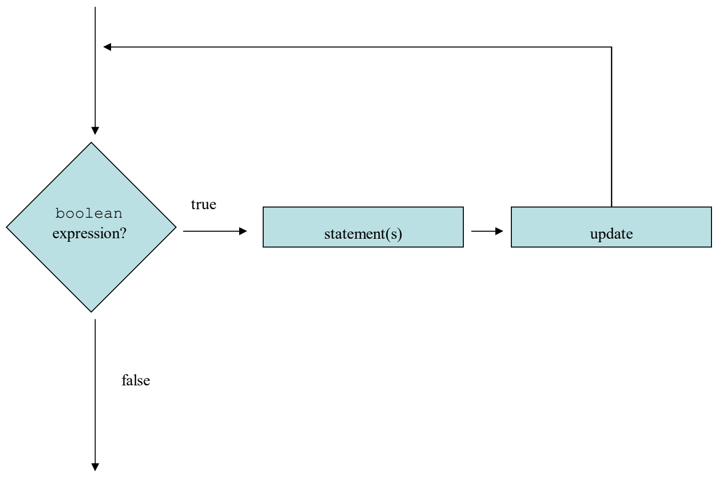
- The for loop is a pre-test loop.
- The for loop allows the programmer to initialize a control variable, test a condition, and modify the control variable all in one line of code.
- The for loop takes the form:

```
for(initialization; test; update)
{
   statement(s);
}
```

- See example: Squares.java

70

The `for` Loop Flowchart



71

The Sections of The `for` Loop

- The *initialization section* of the `for` loop allows the loop to initialize its own control variable.
- The *test section* of the `for` statement acts in the same manner as the condition section of a `while` loop.
- The *update section* of the `for` loop is the last thing to execute at the end of each loop.
- Example: UserSquares.java

72

The `for` Loop Initialization

- The initialization section of a `for` loop is optional; however, it is usually provided.
- Typically, `for` loops initialize a counter variable that will be tested by the test section of the loop and updated by the update section.
- The initialization section can initialize multiple variables.
- Variables declared in this section have scope only for the `for` loop.

73

The Update Expression

- The update expression is usually used to increment or decrement the counter variable(s) declared in the initialization section of the for loop.
- The update section of the loop executes last in the loop.
- The update section may update multiple variables.
- Each variable updated is executed as if it were on a line by itself.

74

Modifying The Control Variable

- You should avoid updating the control variable of a `for` loop within the body of the loop.
- The update section should be used to update the control variable.
- Updating the control variable in the `for` loop body leads to hard to maintain code and difficult debugging.

75

Multiple Initializations and Updates

- The `for` loop may initialize and update multiple variables.

```
for(int i = 5, int j = 0; i < 10 || j < 20; i++,
    j+=2)
{
   statement(s);
}
```

- Note that the only parts of a `for` loop that are mandatory are the semicolons.

```
for(;;)
{
   statement(s);
} // infinite loop
```

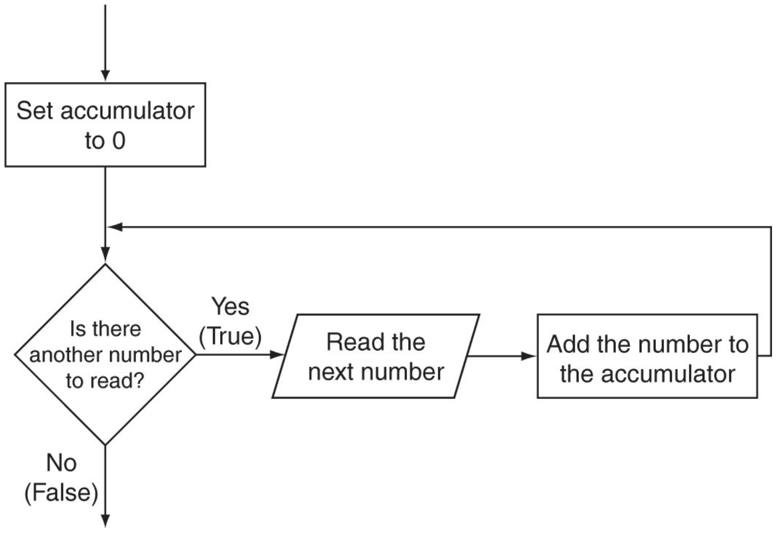
76

Running Totals

- Loops allow the program to keep running totals while evaluating data.
- Imagine needing to keep a running total of user input.
- Example: TotalSales.java

77

Logic for Calculating a Running Total


Set accumulator to 0
Is there another number to read?
Yes (True)
Read the next number
Add the number to the accumulator
No (False)

78

Sentinel Values

- Sometimes the end point of input data is not known.
- A *sentinel value* can be used to notify the program to stop acquiring input.
- If it is a user input, the user could be prompted to input data that is not normally in the input data range (i.e. –1 where normal input would be positive.)
- Programs that get file input typically use the end-of-file marker to stop acquiring input data.
- Example: SoccerPoints.java

79

Nested Loops

- Like `if` statements, loops can be nested.
- If a loop is nested, the inner loop will execute all of its iterations for each time the outer loop executes once.

```
for(int i = 0; i < 10; i++)
    for(int j = 0; j < 10; j++)
        loop statements;
```

- The loop statements in this example will execute 100 times.
- Example: Clock.java

80

The `break` Statement

- The `break` statement can be used to abnormally terminate a loop.
- The use of the `break` statement in loops bypasses the normal mechanisms and makes the code hard to read and maintain.
- It is considered bad form to use the `break` statement in this manner.

81

The `continue` Statement

- The `continue` statement will cause the currently executing iteration of a loop to terminate and the next iteration will begin.
- The `continue` statement will cause the evaluation of the condition in `while` and `for` loops.
- Like the `break` statement, the `continue` statement should be avoided because it makes the code hard to read and debug.

82

Deciding Which Loops to Use

- The `while` loop:
  - Pretest loop
  - Use it where you do not want the statements to execute if the condition is false in the beginning.
- The `do-while` loop:
  - Post-test loop
  - Use it where you want the statements to execute at least one time.
- The `for` loop:
  - Pretest loop
  - Use it where there is some type of counting variable that can be evaluated.

Pearson

83

Some Methods of the `Random` Class

Method	Description
`nextDouble()`	Returns the next random number as a `double`. The number will be within the range of 0.0 and 1.0.
`nextFloat()`	Returns the next random number as a `float`. The number will be within the range of 0.0 and 1.0.
`nextInt()`	Returns the next random number as an `int`. The number will be within the range of an `int`, which is –2,147,483,648 to +2,147,483,648.
`nextInt(int n)`	This method accepts an integer argument, n. It returns a random number as an `int`. The number will be within the range of 0 to n.

See example: RollDice.java

Pearson

84

Backup Slides

85