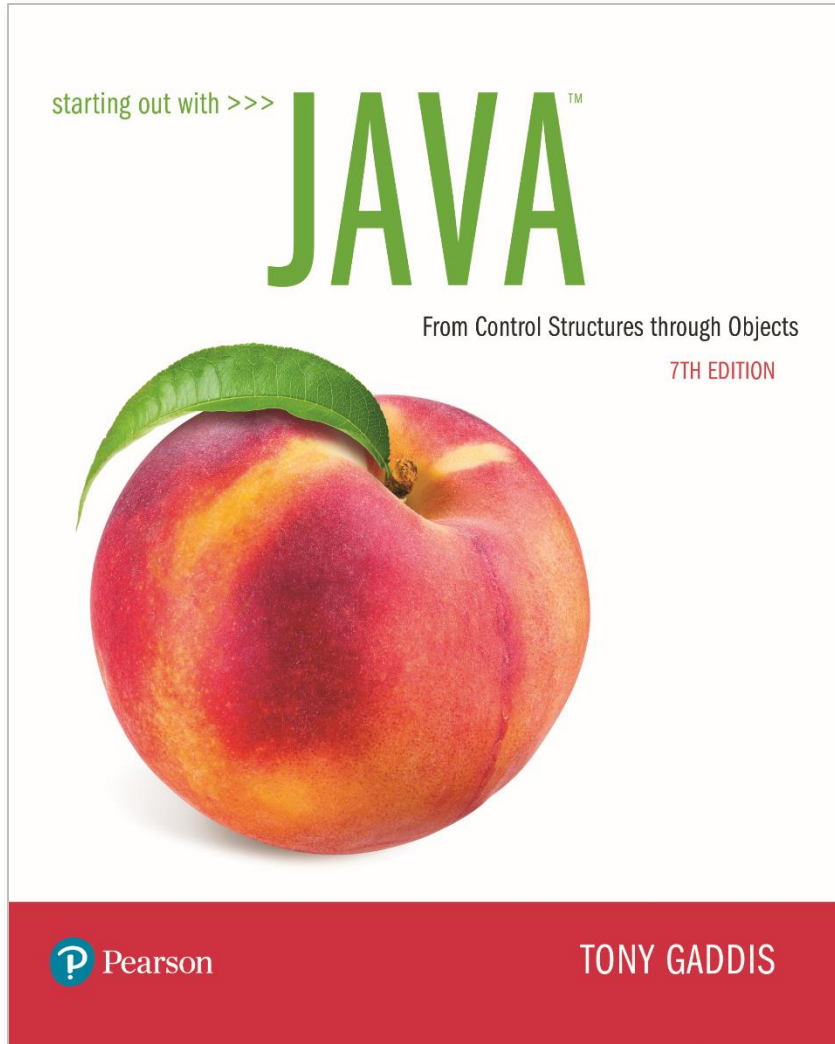


STARTING OUT WITH JAVA™

7th Edition



Chapter 3

Decision Structures

Chapter Topics (1 of 2)

Chapter 3 discusses the following main topics:

- The `if` Statement
- The `if-else` Statement
- Nested `if` statements
- The `if-else-if` Statement
- Logical Operators
- Comparing `String` Objects

Chapter Topics (2 of 2)

Chapter 3 discusses the following main topics:

- More about Variable Declaration and Scope
- The Conditional Operator
- The `switch` Statement
- Displaying Formatted Output with
`System.out.printf` and `String.format`

The `if` Statement

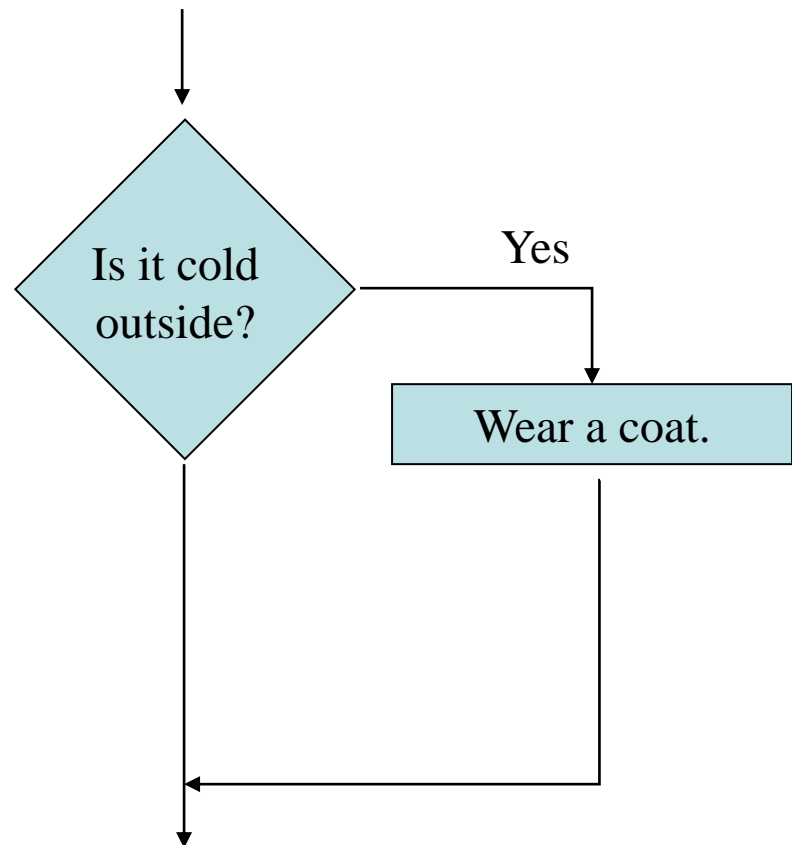
- The `if` statement decides whether a section of code executes or not.
- The `if` statement uses a `boolean` to decide whether the next statement or block of statements executes.

*if (boolean expression is true)
execute next statement.*

Flowcharts (1 of 2)

- If statements can be modeled as a flow chart.

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

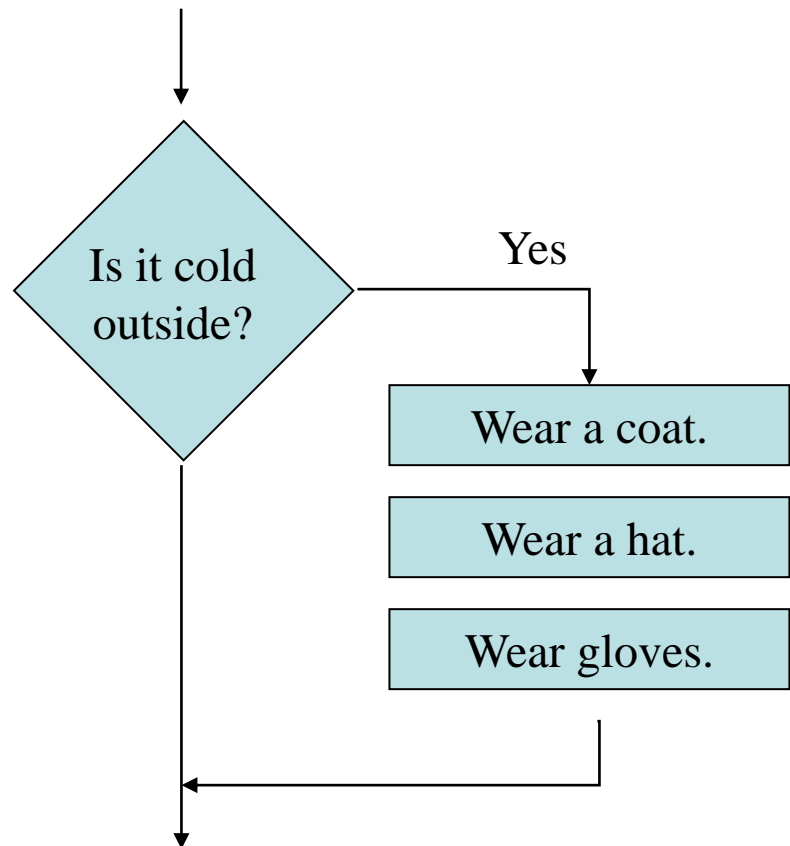


Flowcharts (2 of 2)

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

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

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



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

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?
$x \geq y$	Is x greater than or equal to y?
$x \leq y$	Is x less than or equal to y.
$x == y$	Is x equal to y?
$x != y$	Is x not equal to y?

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](#)

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';
```

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.
```

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.**

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;
```

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!");
```

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");
```

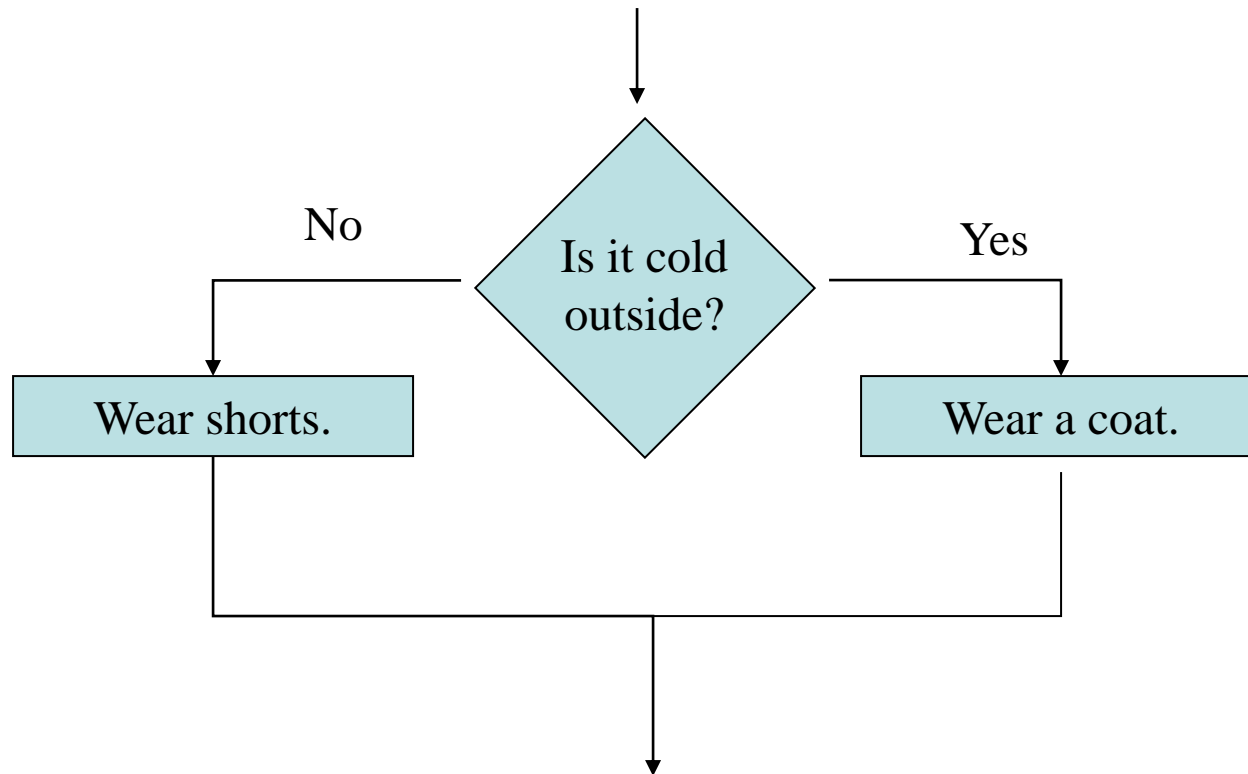
if-else Statements

- 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](#)

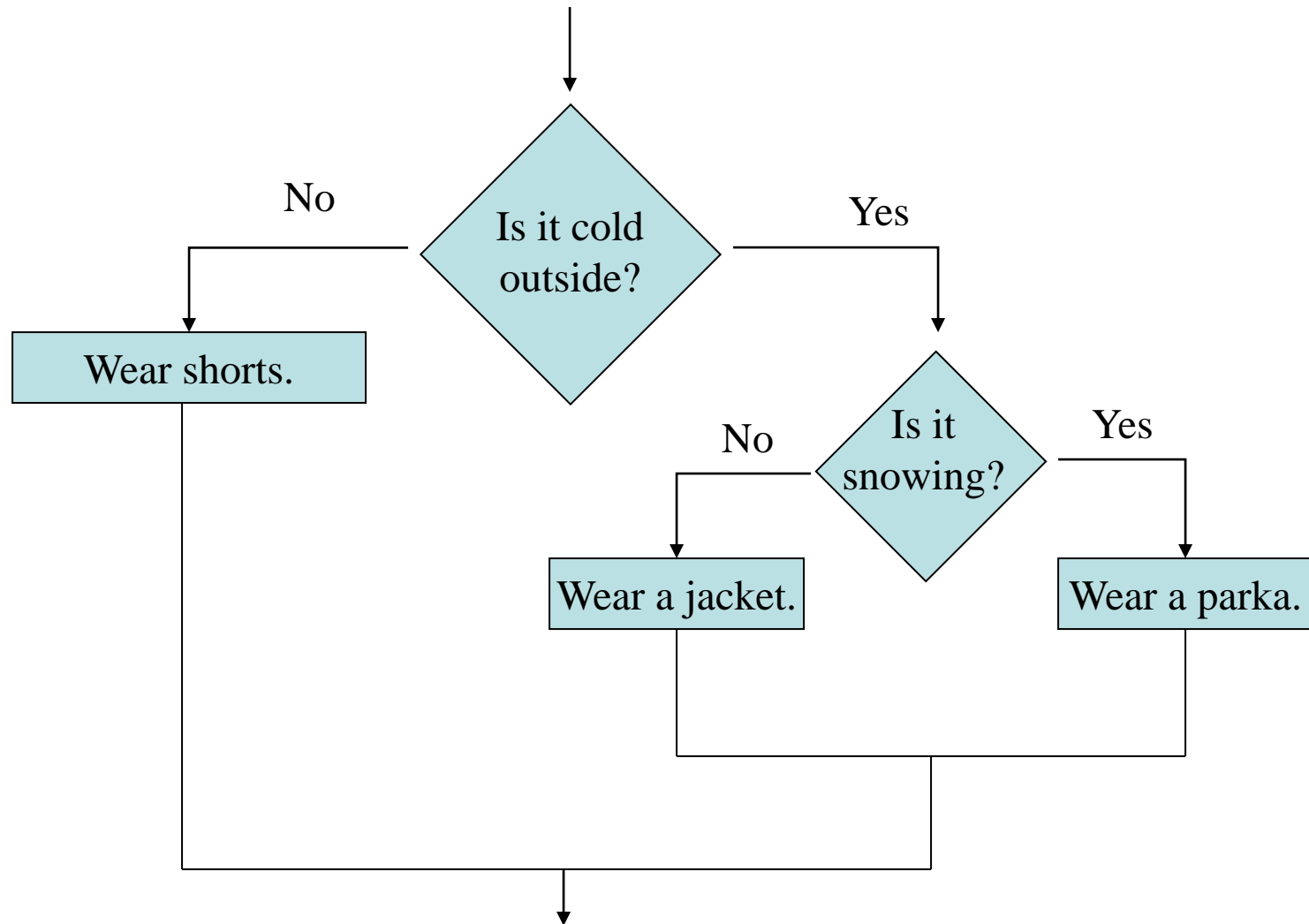
if-else Statement Flowcharts



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](#)

Nested if Statement Flowcharts



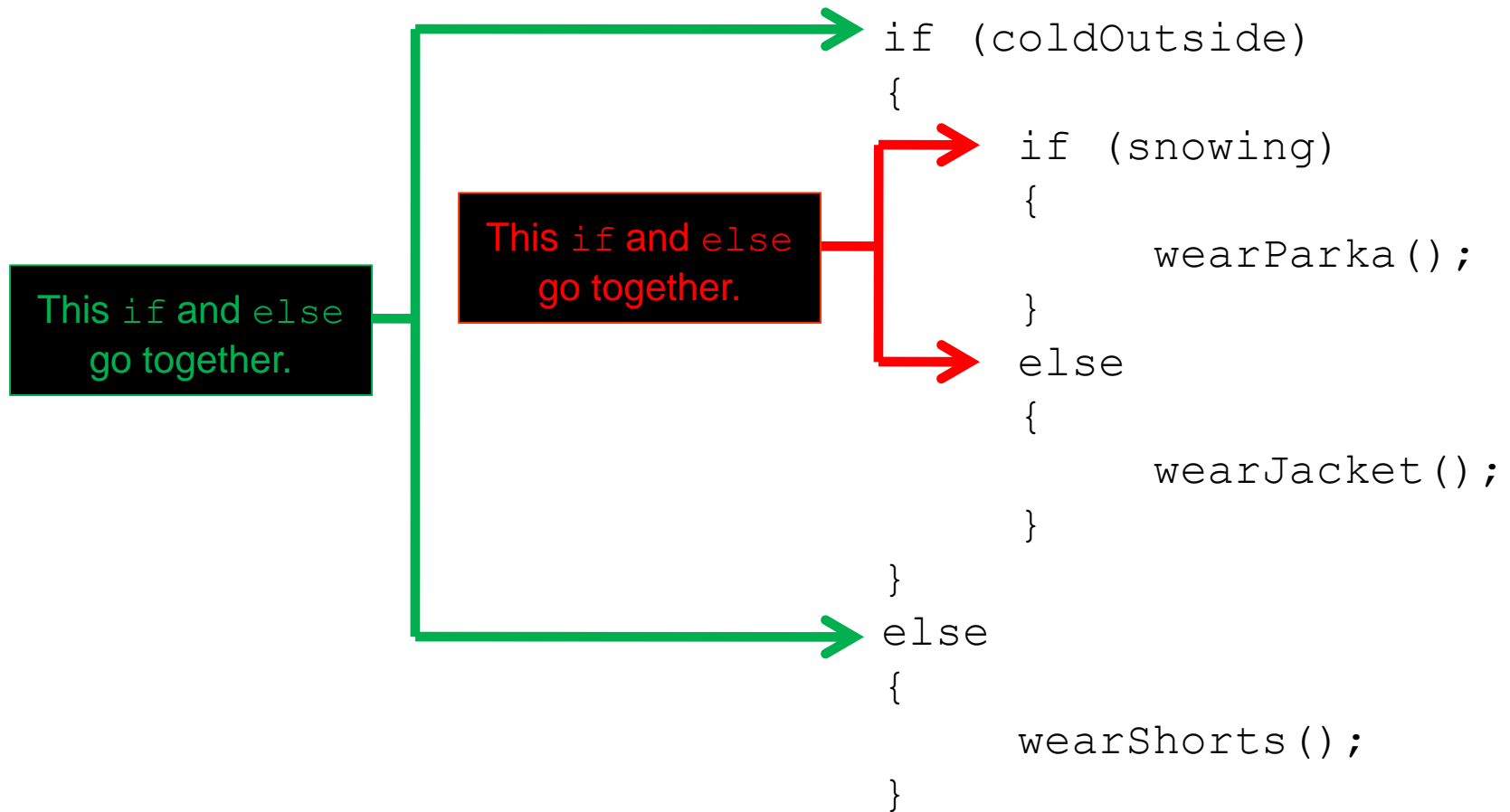
Nested if Statements

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

`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.

Alignment and Nested if Statements



if-else-if Statements (1 of 2)

```
if (expression_1)
{
    statement;
    statement;
    etc.
}
else if (expression_2)
{
    statement;
    statement;
    etc.
}
```

If expression_1 is true these statements are executed, and the rest of the structure is ignored.

Otherwise, if expression_2 is true these statements are executed, and the rest of the structure is ignored.

Insert as many else if clauses as necessary

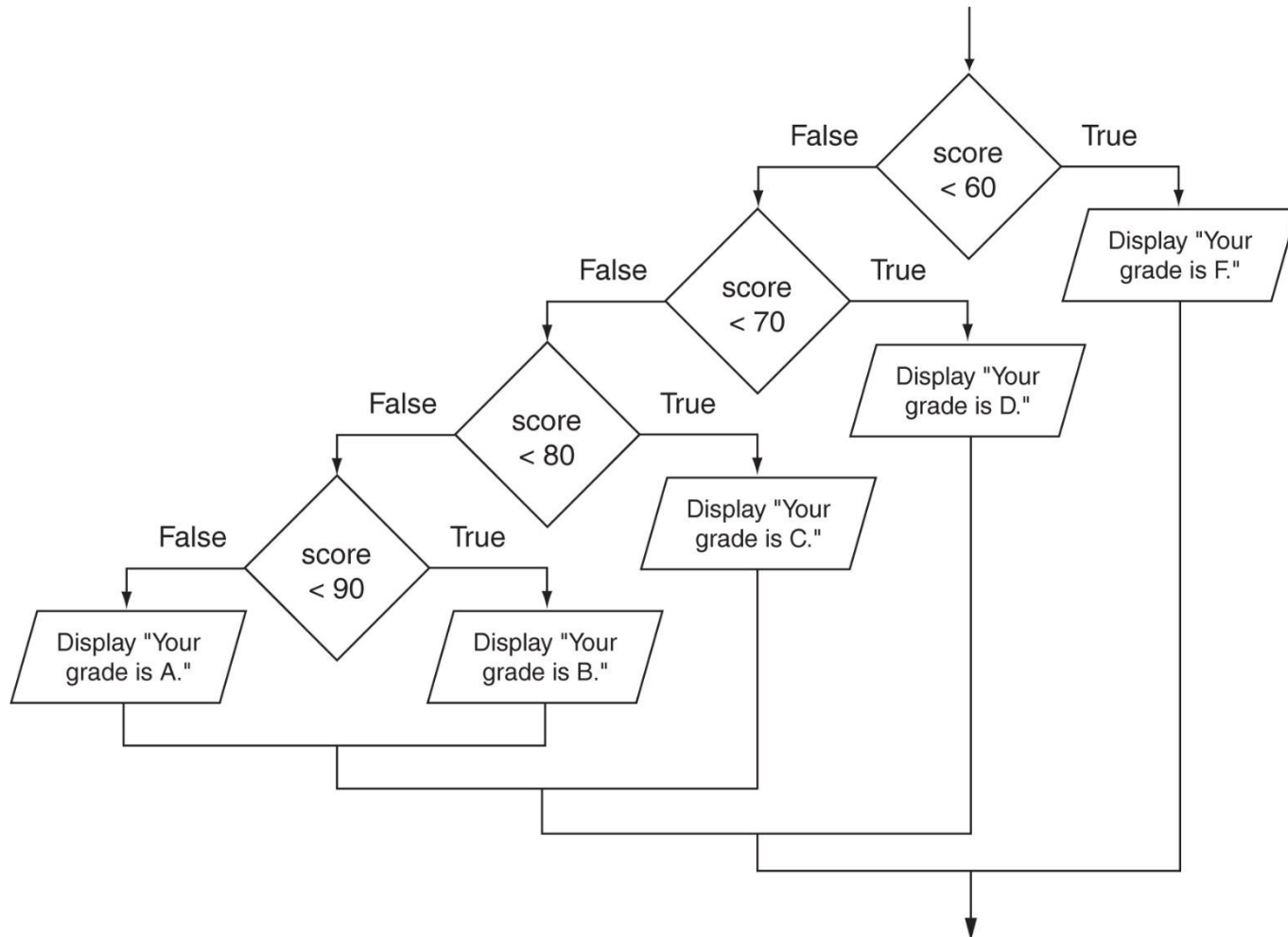
```
else
{
    statement;
    statement;
    etc.
}
```

These statements are executed if none of the expressions above are true.

`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](#)

if-else-if Flowchart



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.

Logical Operators (2 of 2)

Operator	Meaning	Effect
&&	AND	Connects two <code>boolean</code> expressions into one. Both expressions must be true for the overall expression to be true.
 	OR	Connects two <code>boolean</code> 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 <code>boolean</code> 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.

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

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

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

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.

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.

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.

Comparing String Objects

- In most cases, you cannot use the relational operators to compare two `String` objects.
- Reference variables contain the address of the object they represent.
- Unless the references point to the same object, the relational operators will not return true.
- See example: [StringCompare.java](#)
- See example: [StringCompareTo.java](#)

Ignoring Case in String Comparisons

- In the `String` class the `equals` and `compareTo` methods are case sensitive.
- In order to compare two `String` objects that might have different case, use:
 - `equalsIgnoreCase`, or
 - `compareToIgnoreCase`
- See example: [SecretWord.java](#)

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.

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*.

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;
```

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](#)

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.

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
}
```

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.

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.

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](#)

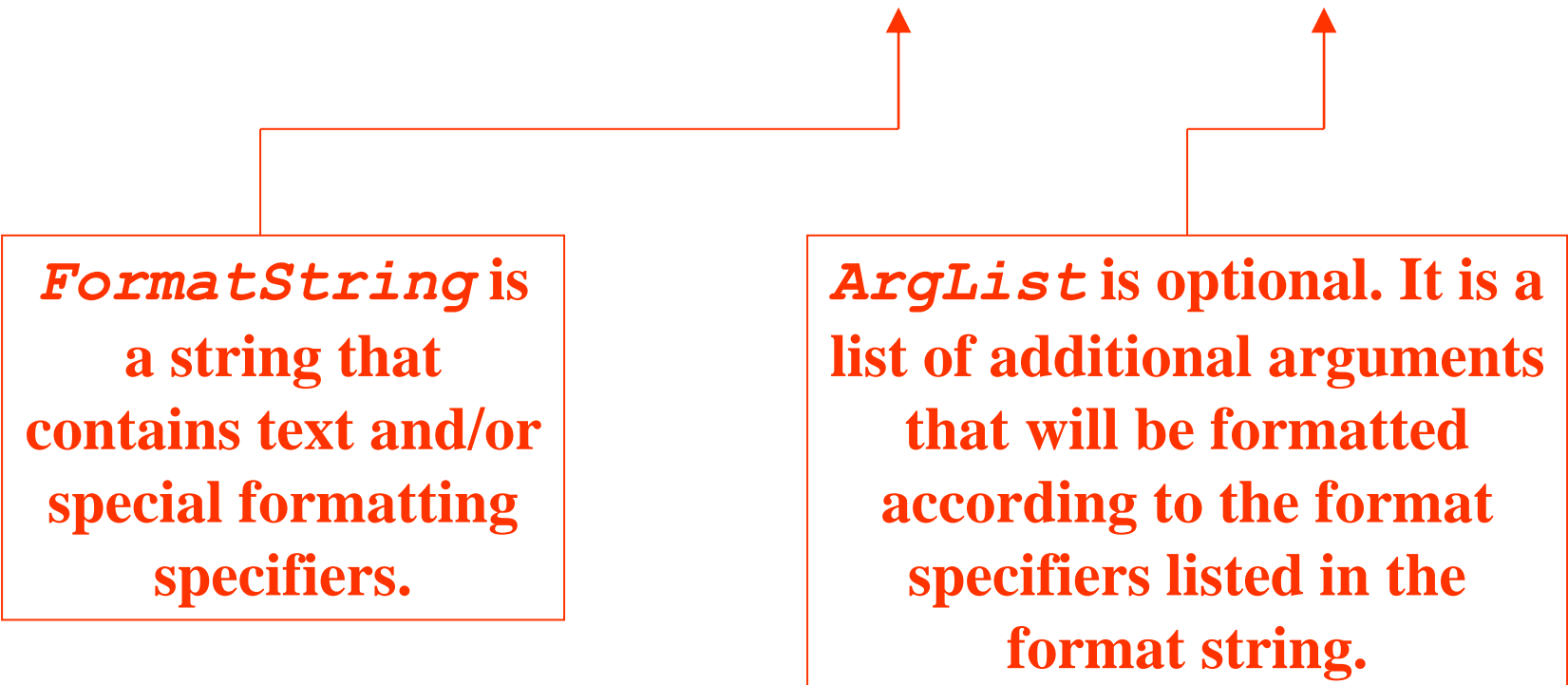
The `System.out.printf` Method (1 of 15)

- 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);
```

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

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



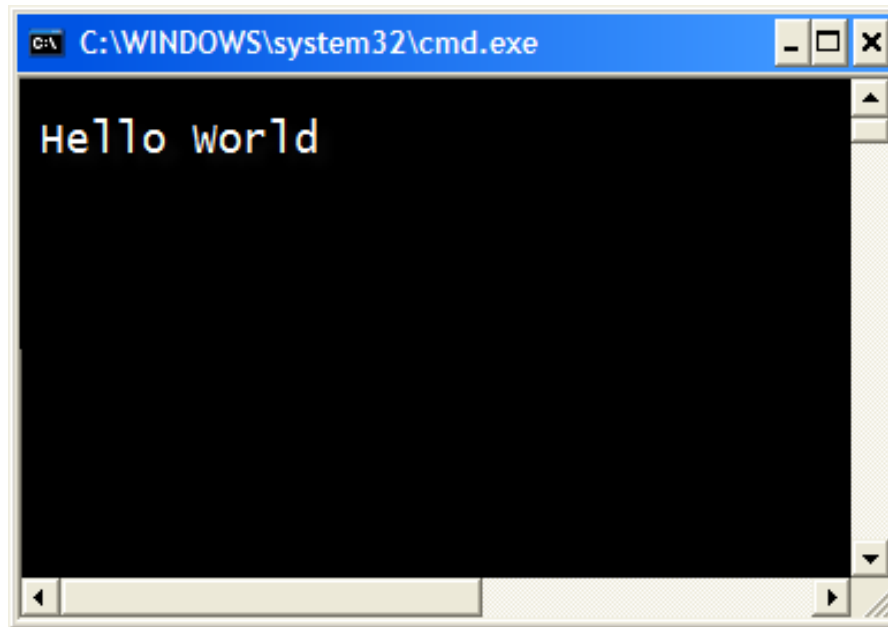
FormatString is a string that contains text and/or special formatting specifiers.

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

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

- A simple example:

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

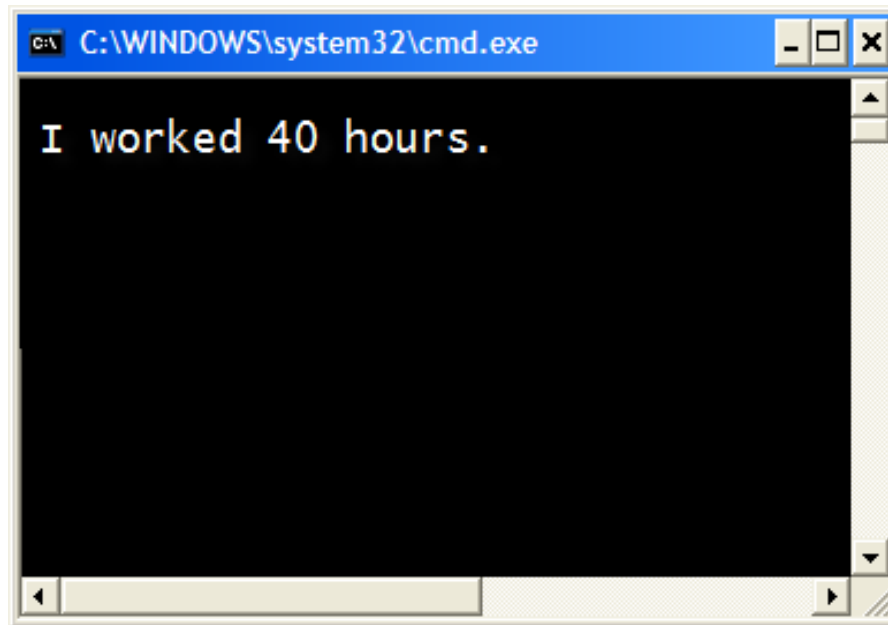


The `System.out.printf` Method (4 of 15)

- Another example:

```
int hours = 40;
```

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



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

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

The diagram illustrates the data flow in the provided code. A red circle highlights the `%d` format specifier in the `printf` statement. Two red arrows originate from this circle: one points to a box explaining the meaning of `%d`, and the other points to a box explaining that the value from the `hours` variable is passed to it. Additionally, a red arrow points from the `hours` variable in the first line to the `%d` specifier in the second line.

The `%d` format specifier indicates that a decimal integer will be printed.

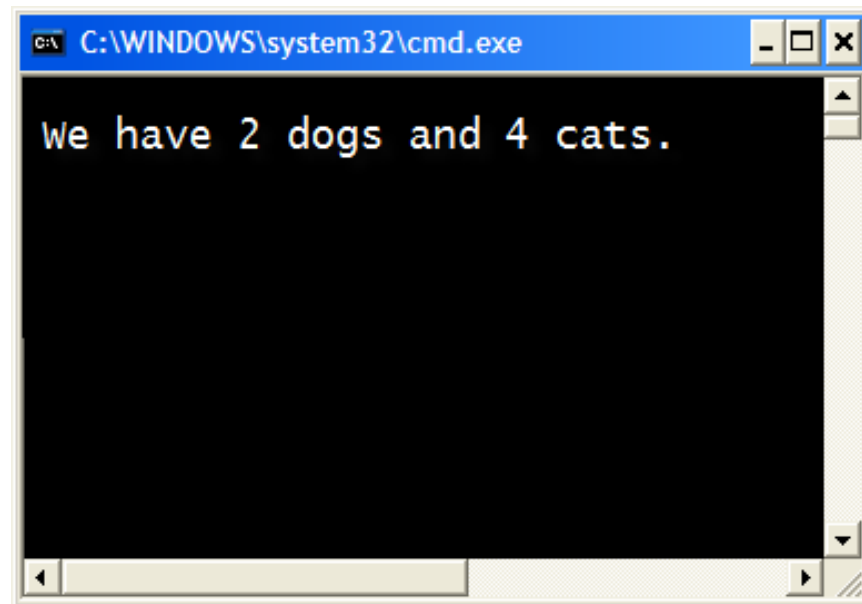
The contents of the `hours` variable will be printed in the location of the `%d` format specifier.

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);
```

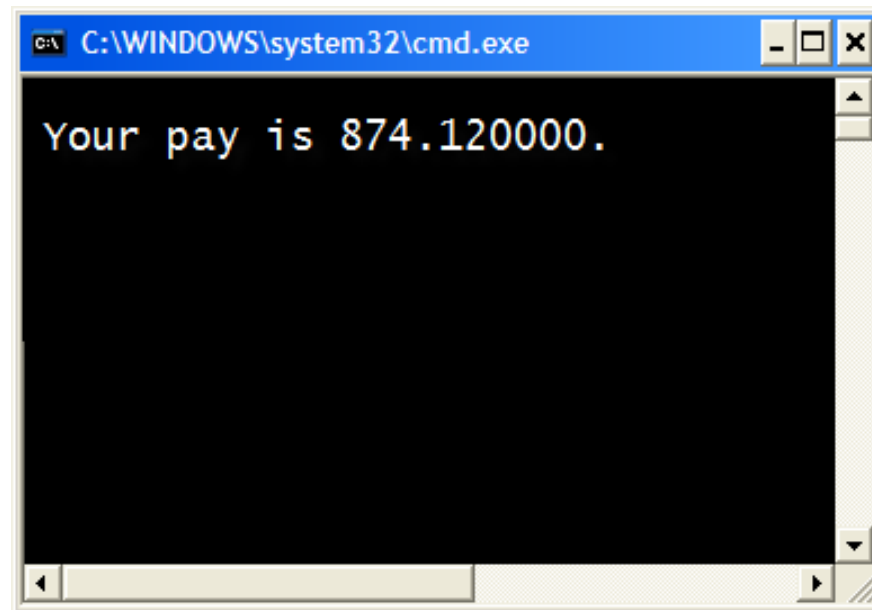


The `System.out.printf` Method (7 of 15)

- Another example:

```
double grossPay = 874.12;
```

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



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

- Another example:

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

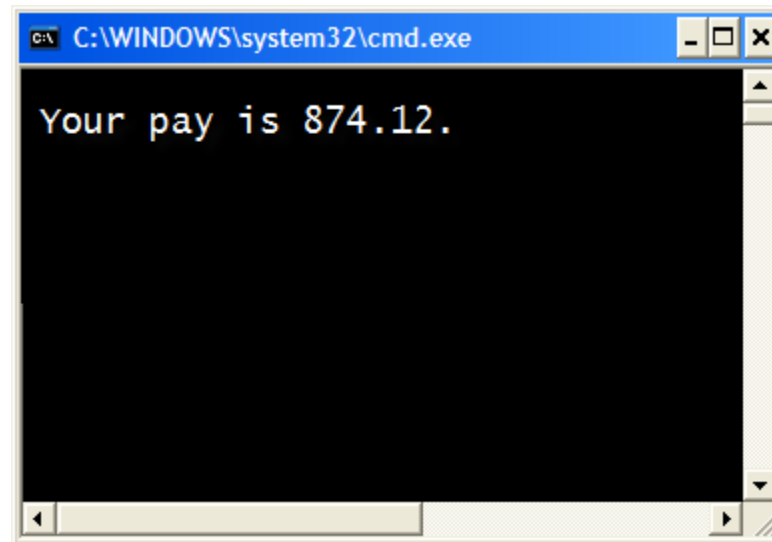
The `%f` format specifier indicates that a floating-point value will be printed.

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

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

- Another example:

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




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.

The System.out.printf Method

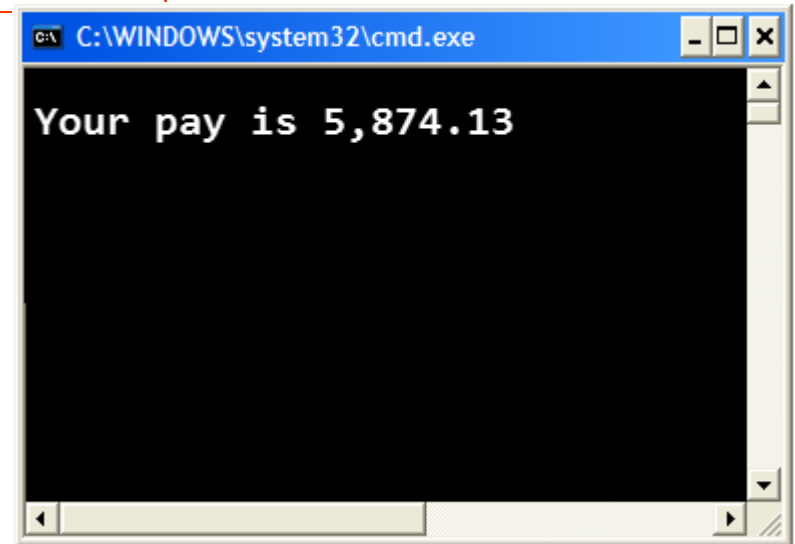
(11 of 15)

- 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.

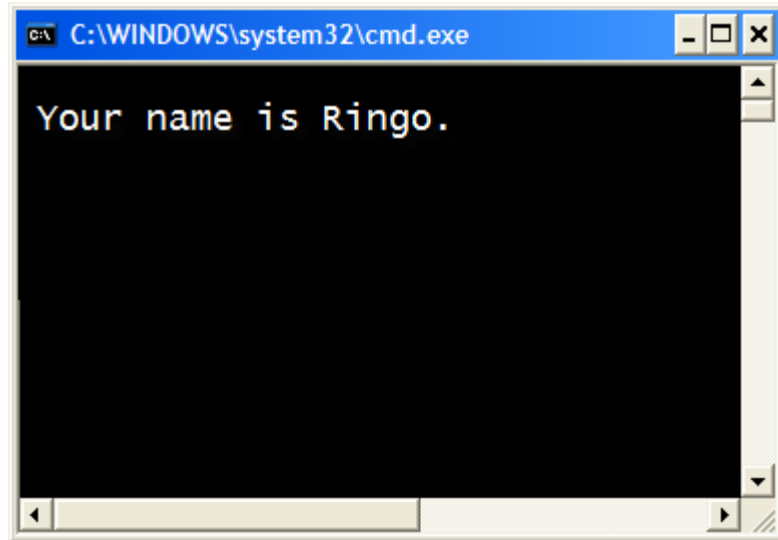


The System.out.printf Method (12 of 15)

- Another example:

```
String name = "Ringo";
```

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

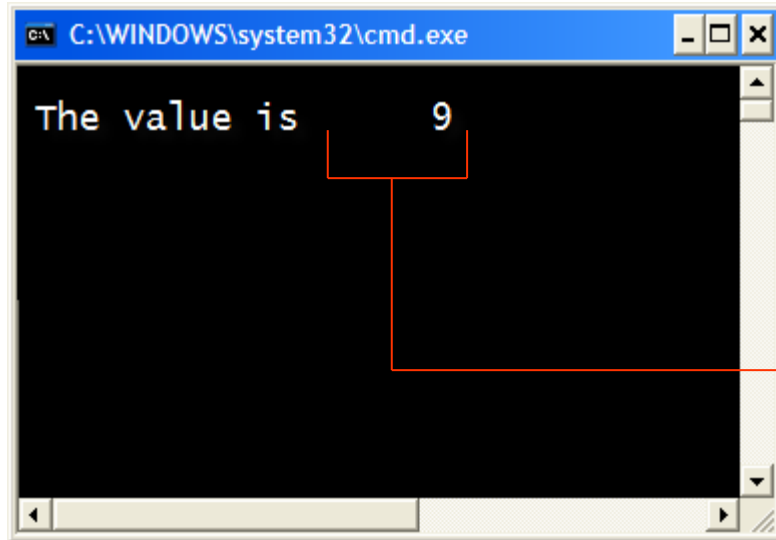


The %s format specifier indicates that a string will be printed.

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

- Specifying a field width:

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



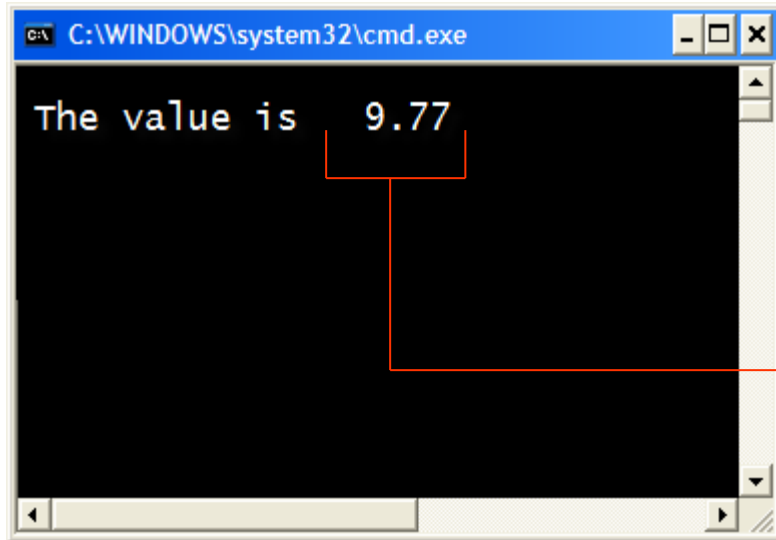
The %6d format specifier indicates the integer will appear in a field that is 6 spaces wide.

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 %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](#)

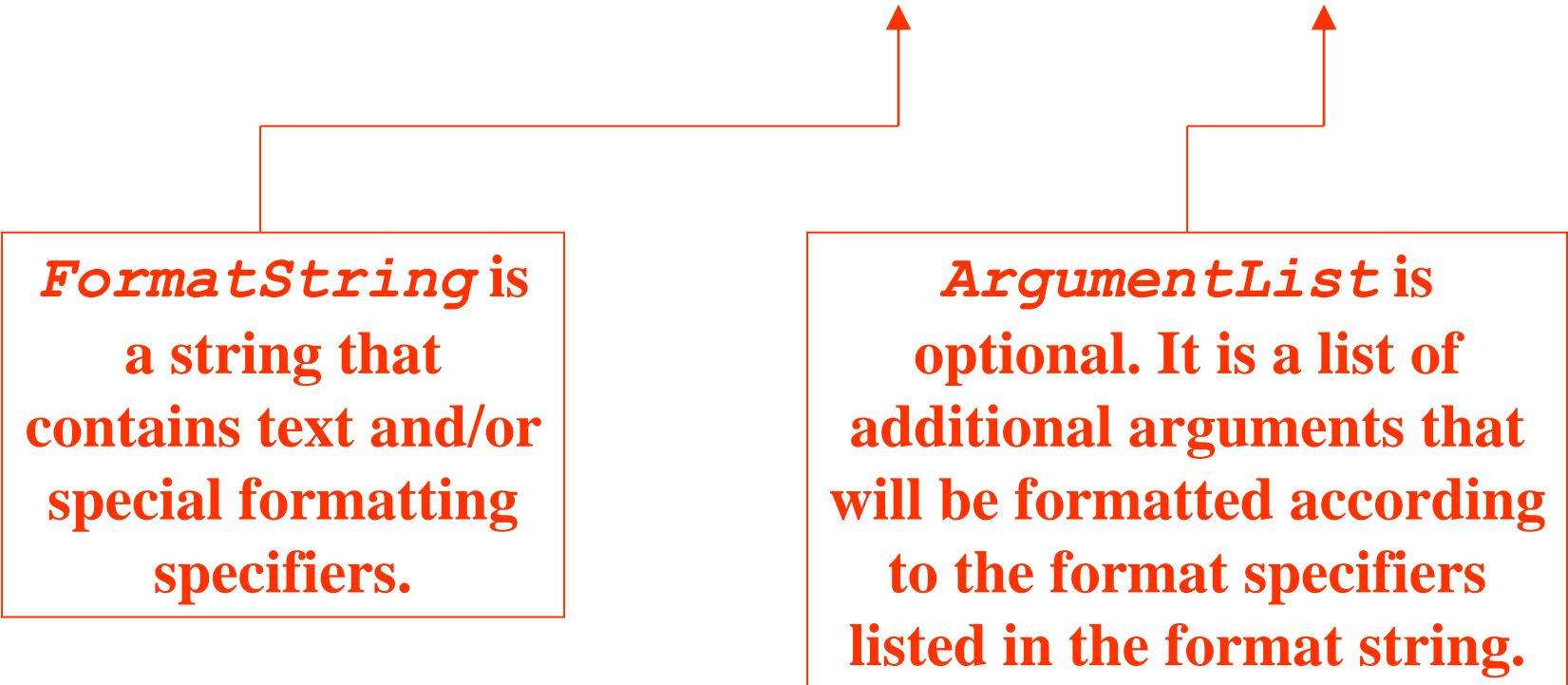
The `String.format` Method (1 of 3)

- The `String.format` method works exactly like the `System.out.printf` method, except that it does not display the formatted string on the screen.
- Instead, it returns a reference to the formatted string.
- You can assign the reference to a variable, and then use it later.

The `String.format` Method (2 of 3)

- The general format of the method is:

```
String.format(FormatString, ArgumentList);
```



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

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

The `String.format` Method (3 of 3)

- See examples:
 - [CurrencyFormat2.java](#)
 - [CurrencyFormat3.java](#)

Copyright



This work is protected by United States copyright laws and is provided solely for the use of instructions in teaching their courses and assessing student learning. dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.