

starting out with >>>

JAVATM **EARLY OBJECTS**

FIFTH EDITION

CHAPTER 4

Decision Structures



TONY GADDIS

Addison-Wesley
is an imprint of

PEARSON

Copyright © 2015 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

Topics

- 🍓 The `if` Statement
- 🍓 The `if-else` Statement
- 🍓 The `PayRoll` class
- 🍓 Nested `if` Statements
- 🍓 The `if-else-if` Statement
- 🍓 Logical Operators
- 🍓 Comparing `String` Objects

Topics (cont'd)

- 🍓 More about Variable Declaration and Scope
- 🍓 The Conditional Operator
- 🍓 The `switch` Statement
- 🍓 The `DecimalFormat` Class
- 🍓 The `SalesCommission` Class
- 🍓 Generating Random Numbers with the `Random` Class

The `if` Statement

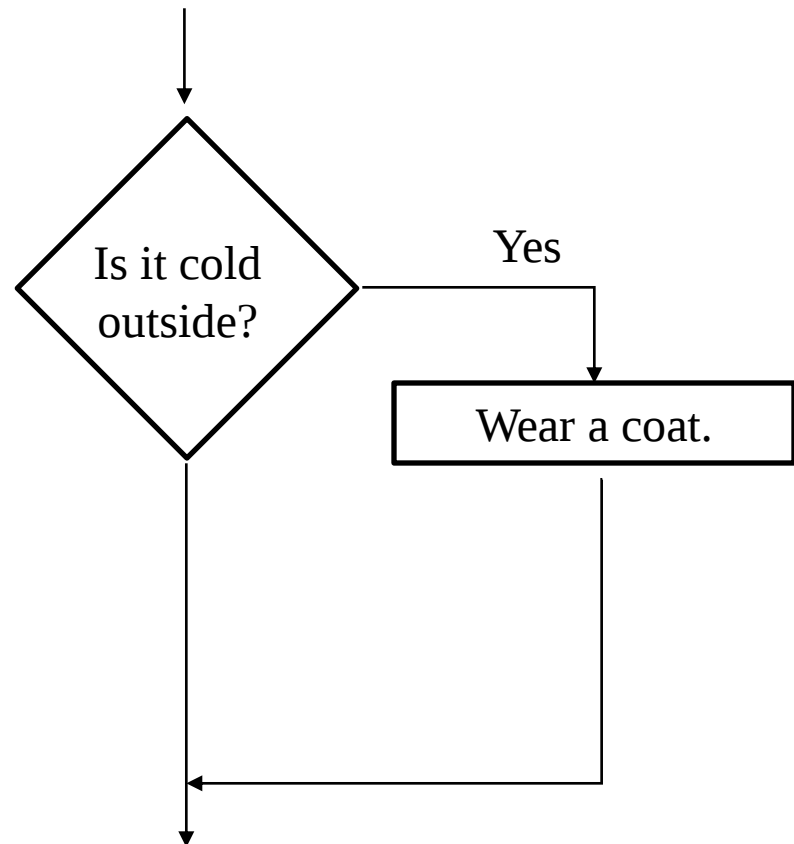
- 🍷 The `if` statement decides whether a section of code executes or not.
- 🍷 The `if` statement uses a `boolean` to decide whether the next statement or block of statements executes.

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

Flowcharts

- If statements can be modeled as a flow chart.

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

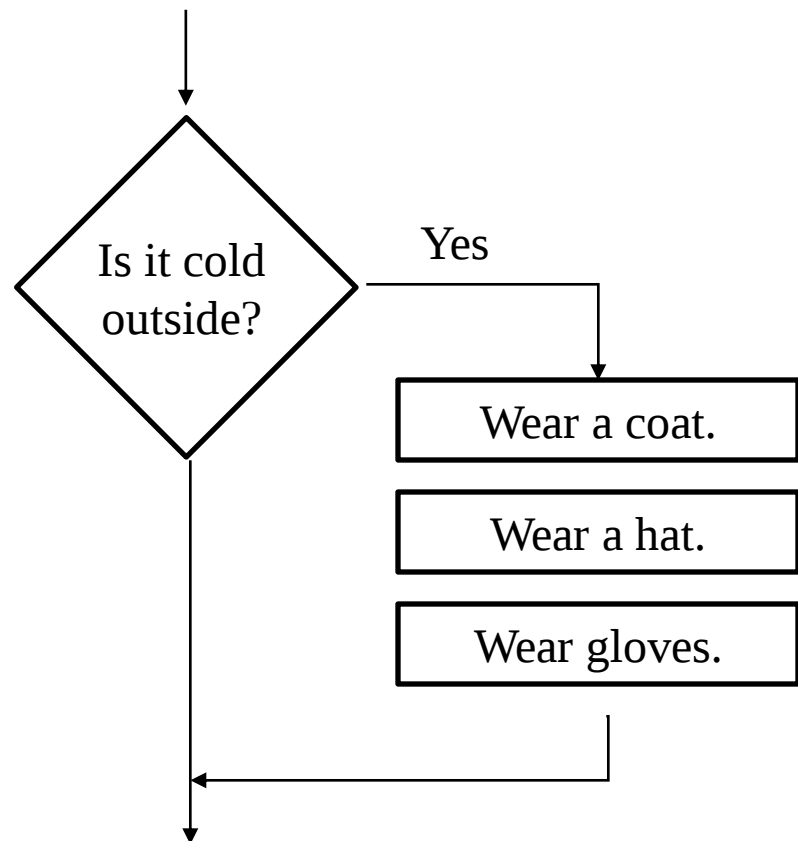


Flowcharts (cont'd)

- 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

- 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 (cont'd)

- **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.
```

Having Multiple Conditionally-Executed Statements

- 🍓 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.
```

Having Multiple Conditionally-Executed Statements (cont'd)

- 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

- ❶ Characters can be tested using the relational operators.
- ❷ Characters are stored in the computer using the Unicode character format.
- ❸ Unicode is stored as a sixteen (16) bit number.
- ❹ Characters are *ordinal*, meaning they have an order in the Unicode character set.
- ❺ 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.

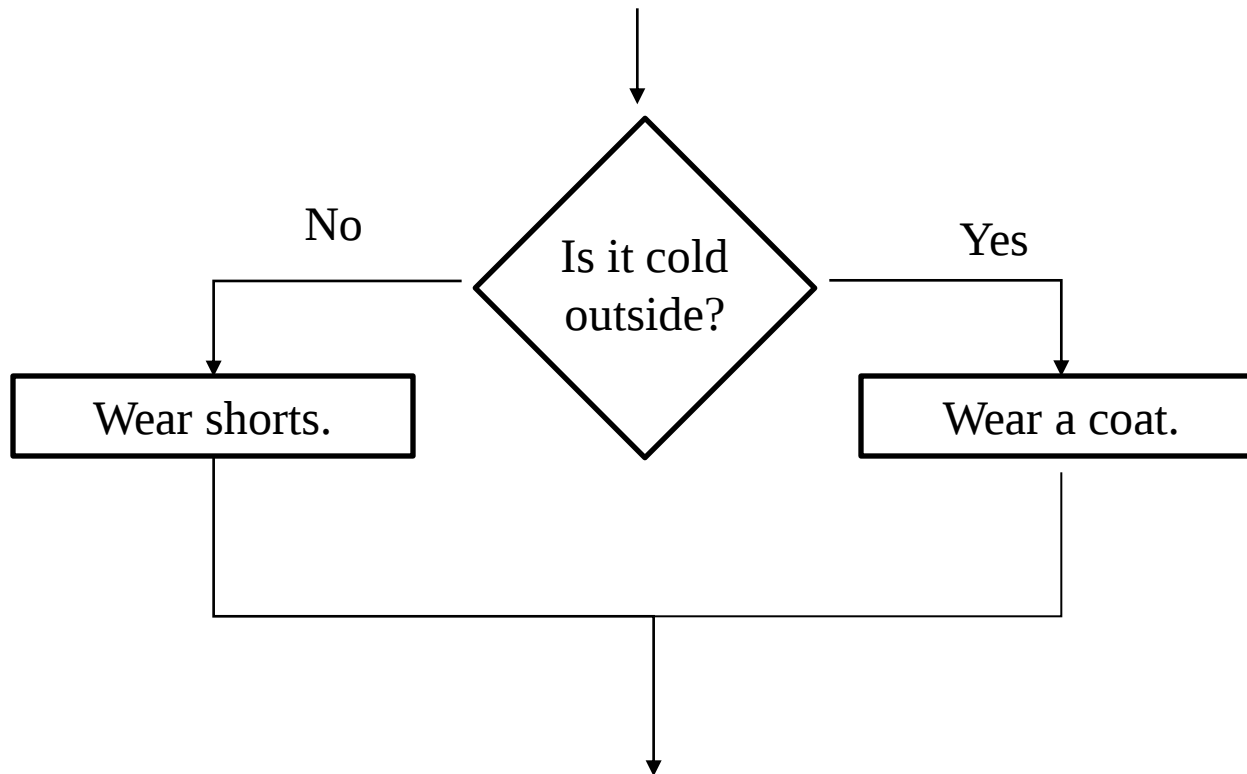
```
if (expression)  
    statementOrBlockIfTrue;  
else  
    statementOrBlockIfFalse;
```

- 🍓 See example: [Division.java](#)

The Payroll Class

Payroll
- hoursWorked : double - payRate : double
+ Payroll() + setHoursWorked(hours : double) : void + setPayRate(rate : double): void + getHoursWorked() : double + getPayRate() : double + getGrossPay() : double

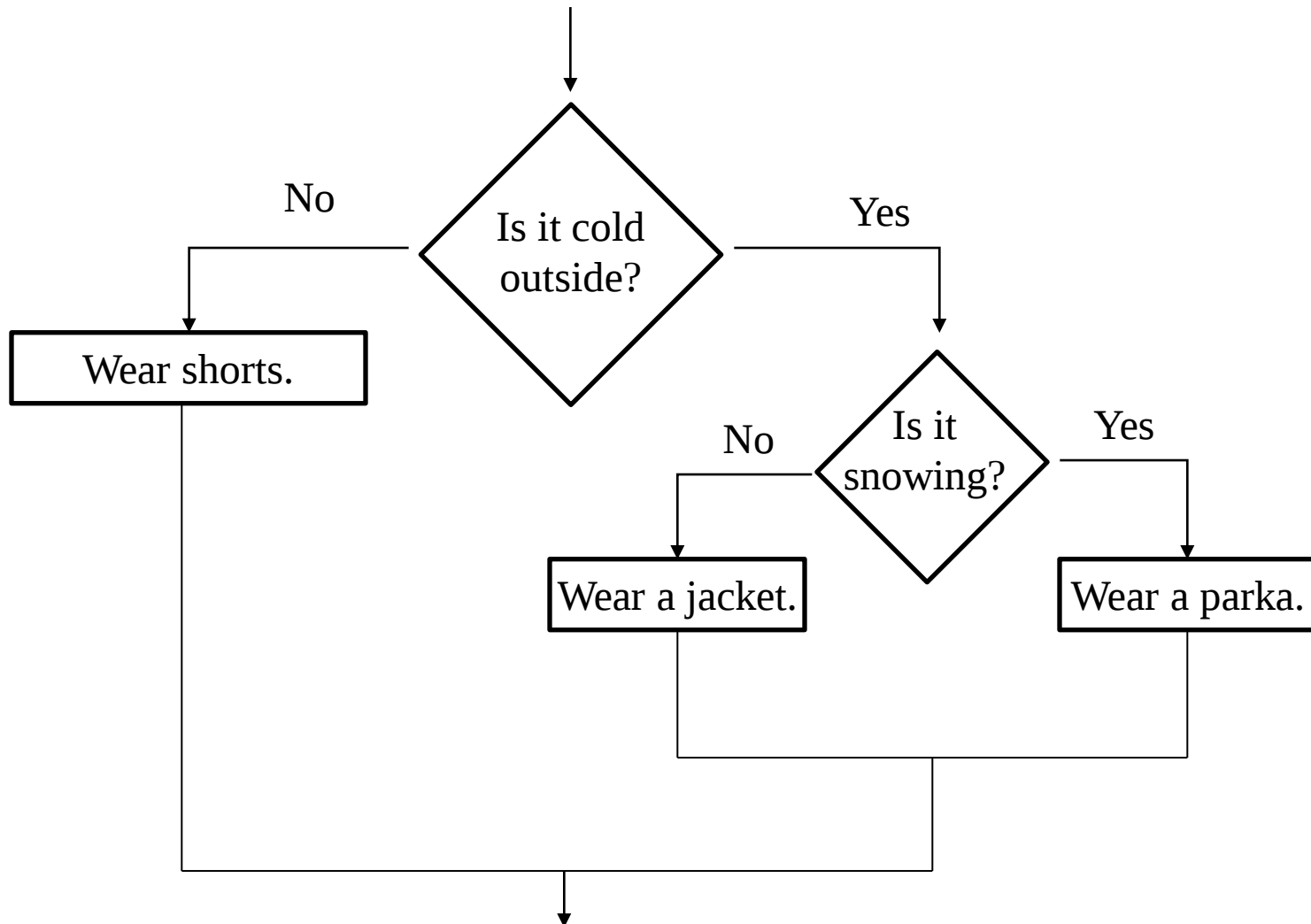
if-else Statement Flowcharts



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



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.**

if-else Matching

```
if (employed == 'y')
{
    if (recentGrad == 'y')
    {
        System.out.println("You qualify for the " +
                           "special interest rate.");
    }
    else
    {
        System.out.println("You must be a recent " +
                           "college graduate to qualify.");
    }
}
else
{
    System.out.println("You must be employed to qualify.");
}
```

if-else-if Statements

🍓 **if-else-if statements can become very complex.**

🍓 **Imagine the following decision set.**

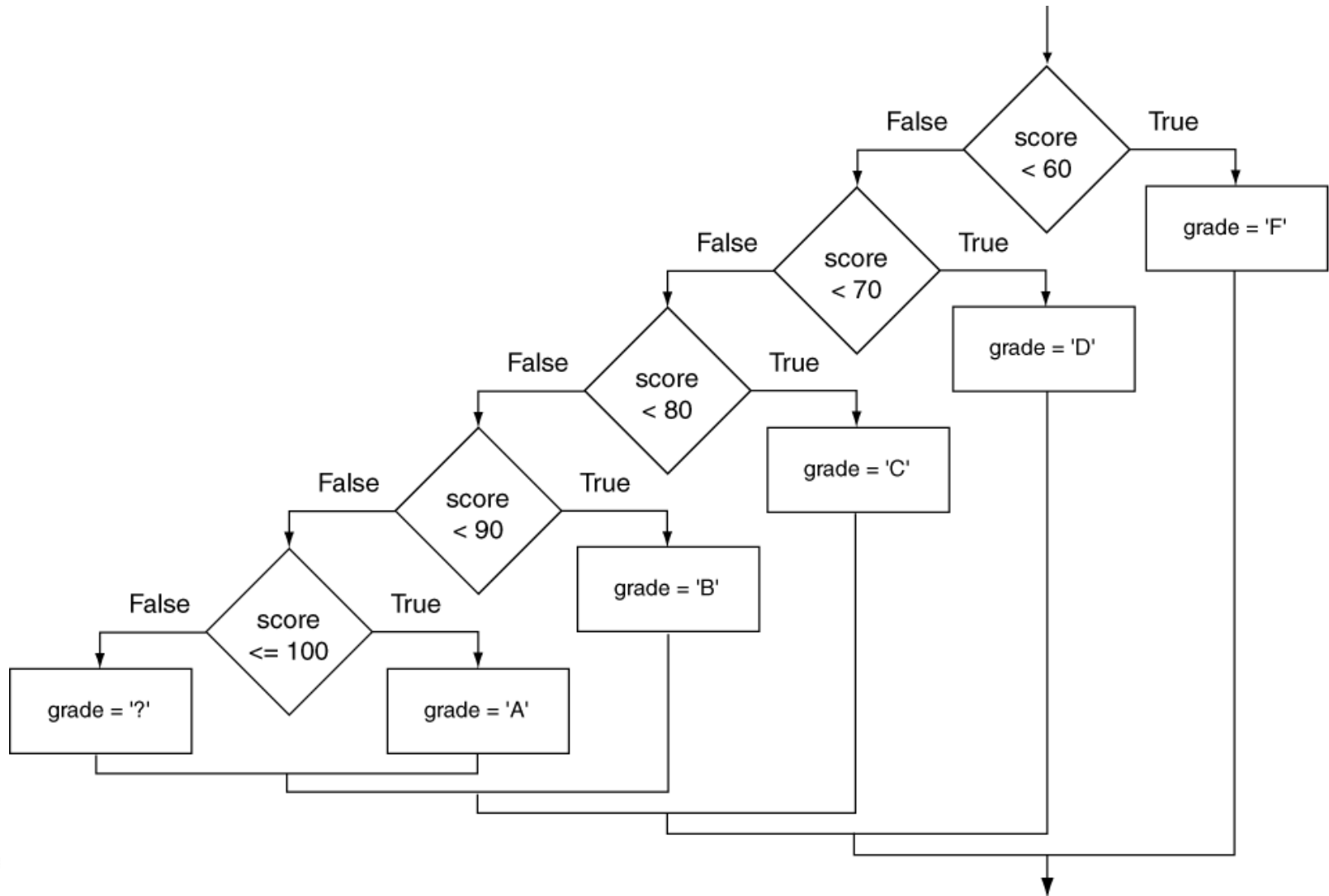
*if it is very cold, wear a heavy coat,
else, if it is chilly, wear a light jacket,
else, if it is windy wear a windbreaker,
else, if it is hot, wear no jacket.*

if-else-if Statements

```
if (expression)  
    statement or block  
else if (expression)  
    statement or block  
    // Put as many else ifs as needed here  
else  
    statement or block
```

- ❗ Care must be used since `else` statements match up with the immediately preceding unmatched `if` statement.
- ❗ See example:
[TestGrade.java](#), [TestResults.java](#)

if-else-if Flowchart



Logical Operators

- 🍓 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

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

- 🍓 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

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: [GoodStringCompare.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`
 - 🍓 `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

- 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 format of the operators is:
$$expression1 \ ? \ expression2 \ : \ expression3$$
- The conditional operator can also return a value.

The Conditional Operator

- 🍷 The conditional operator can be used as a shortened `if-else` statement:

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

- 🍷 This line is functionally equivalent to:

```
if (x > y)
```

```
    z = 10;
```

```
else
```

```
    z = 5;
```

The Conditional Operator

- 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

- 🍷 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 a `char`, `byte`, `short`, `int`, or `string` value and make decisions based on the value.

The switch Statement

 The switch statement takes the form:

```
switch (testExpression)
{
    case Value_1:
        // place one or more statements here
        break;
    case Value_2:
        // 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

- The *testExpression* is a variable or expression that gives a char, byte, short, int or string value.

```
switch (testExpression)  
{  
    ...  
}
```

- The switch statement will evaluate the *testExpression*.
- If there is an associated case statement that matches that value, program execution will be transferred to that case statement.

The switch Statement

- 🍷 Each case statement will have a corresponding case value that must be unique.

```
case value_1:  
    // place one or more statements here  
    break;
```

- 🍷 If the *testExpression* matches the case value, 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 DecimalFormat Class

- 🍓 When printing out `double` and `float` values, the full fractional value will be printed.
- 🍓 The `DecimalFormat` class can be used to format these values.
- 🍓 In order to use the `DecimalFormat` class, the following `import` statement must be used at the top of the program:
`import java.text.DecimalFormat;`
- 🍓 See examples:
[Format1.java](#), [Format2.java](#), [Format3.java](#), [Format4.java](#)

The SalesCommission Class

SalesCommision
- sales : double - rate : double - commission : double - advance : double - pay : double
+ SalesCommission(s: double, a: double) : - setRate() : void -calculatePay() : void + getPay() : double + getCommission() : double + getRate() : double + getAdvance() : double + getSales() : double

Method Decomposition

Using Private Methods

- 🍓 Long methods can be broken up into shorter more specialized methods, known as *helper* or *utility* methods.
- 🍓 Each method should perform a small, well-defined task.
- 🍓 Sensitive tasks can be broken into private methods.
- 🍓 Private methods are available only to other methods within the class.

Main Programs

- 🍷 The `SalesCommission` class is a utility class.
- 🍷 It provides an abstraction of the data and methods needed to calculate a sales commission.
- 🍷 The program that utilizes this class is called the Main program.
- 🍷 Example: [HalsCommission.java](#)

Generating Random Numbers with the Random Class

- 🍓 Some applications, such as games and simulations, require the use of randomly generated numbers.
- 🍓 The Java API has a class, `Random`, for this purpose. To use the `Random` class, use the following `import` statement and create an instance of the class.

```
import java.util.Random;
```

```
Random randomNumbers = new Random();
```

Some Methods of the Random Class

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

See examples: [MathTutor.java](#), [DiceDemo.java](#)