# Boolean Expressions

# Outline

- Introduction
- Java Comparison Operators
- Evaluating Boolean Expressions
- Pitfall: Using **==** with Strings
- Lexicographic and Alphabetical Order
- Building Boolean Expressions
- Truth Tables
- Short-Circuit and Complete Evaluation
- Precedence and Associativity Rules
- Evaluating Expressions
- Rules for Evaluating Expressions

# - Introduction

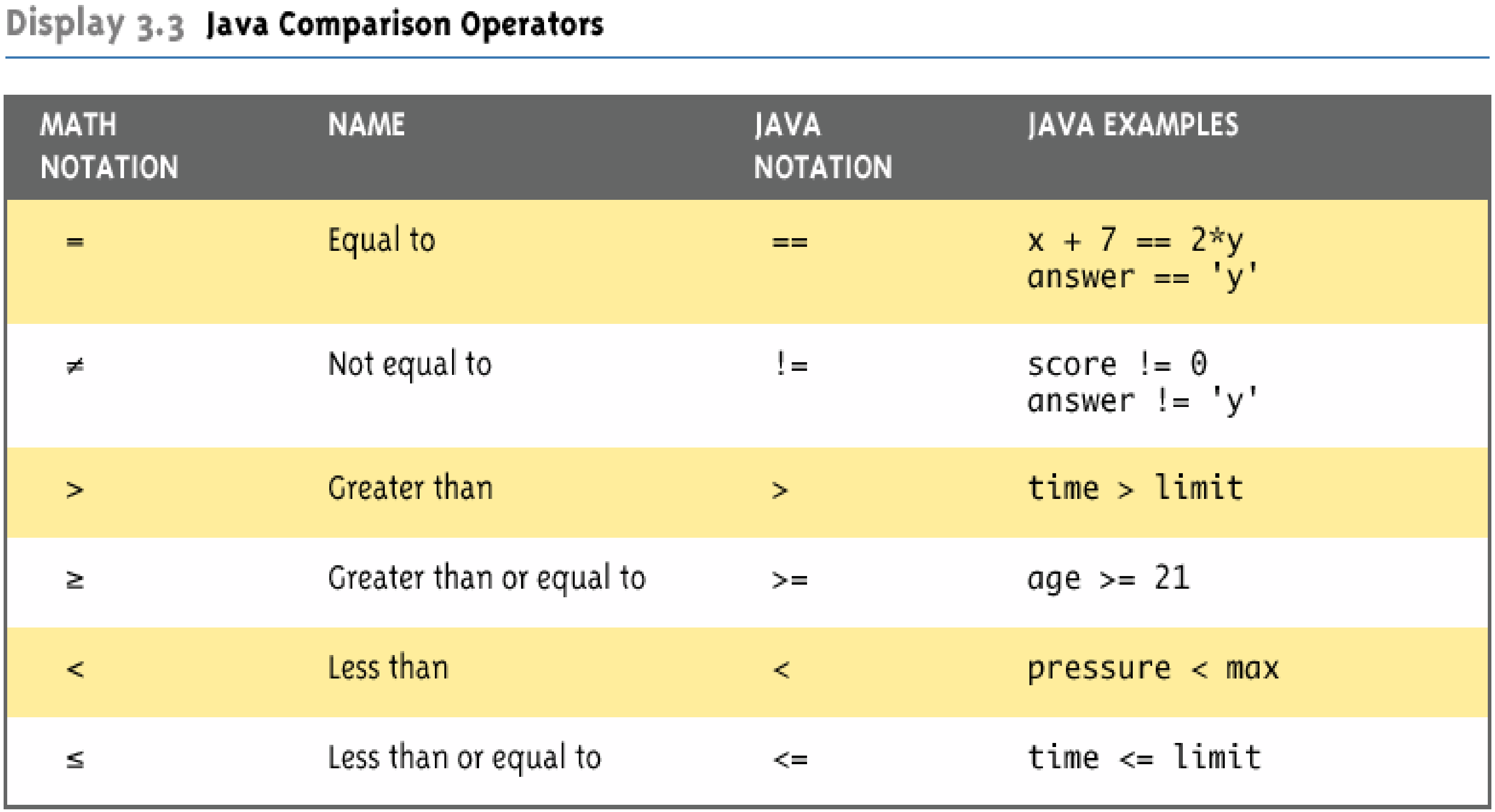
- A Boolean expression is an expression that is either **true** or **false**

- The simplest Boolean expressions compare the value of two expressions

```
time < limit
yourScore == myScore
```

  - Note that Java uses two equal signs (**==**) to perform equality testing:  A single equal sign (**=**) is used only for assignment

# - Java Comparison Operators



Display 3.3 **Java Comparison Operators**

| MATH NOTATION | NAME | JAVA NOTATION | JAVA EXAMPLES |
|---|---|---|---|
| = | Equal to | `==` | `x + 7 == 2*y`<br>`answer == 'y'` |
| ≠ | Not equal to | `!=` | `score != 0`<br>`answer != 'y'` |
| > | Greater than | `>` | `time > limit` |
| ≥ | Greater than or equal to | `>=` | `age >= 21` |
| < | Less than | `<` | `pressure < max` |
| ≤ | Less than or equal to | `<=` | `time <= limit` |

# - Evaluating Boolean Expressions

- Even though Boolean expressions are used to control branch and loop statements, Boolean expressions can exist independently as well
  - A Boolean variable can be given the value of a Boolean expression by using an assignment statement
- A Boolean expression can be evaluated in the same way that an arithmetic expression is evaluated
  - The only difference is that arithmetic expressions produce a number as a result, while Boolean expressions produce either **true** or **false** as their result

```
boolean madeIt = (time < limit) && (limit < max);
```

# - Pitfall: Using == with Strings

- The equality comparison operator (**==**) can correctly test two values of a *primitive* type
- However, when applied to two *objects* such as objects of the **String** class, **==** tests to see if they are stored in the same *memory location*, not whether or not they have the same value
- In order to test two strings to see if they have equal values, use the method **equals**, or **equalsIgnoreCase**

```
string1.equals(string2)
string1.equalsIgnoreCase(string2)
```

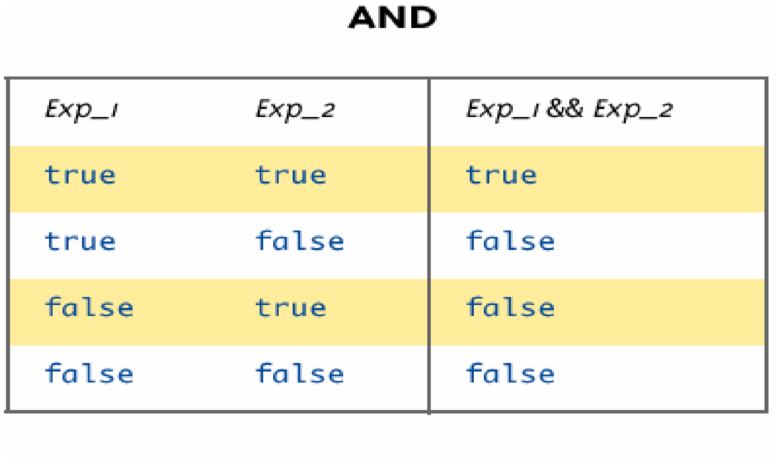
# - Lexicographic and Alphabetical Order

- *Lexicographic* ordering is the same as *ASCII* ordering, and includes letters, numbers, and other characters
  - All uppercase letters are in alphabetic order, and all lowercase letters are in alphabetic order, but all uppercase letters come before lowercase letters
  - If **s1** and **s2** are two variables of type **String** that have been given **String** values, then **s1.compareTo(s2)** returns:
    - A negative number if **s1** is before **s2** in lexicographic ordering
    - zero if the two strings are equal.
    - A positive number if **s2** comes before **s1**
- When performing an alphabetic comparison of strings (rather than a lexicographic comparison) that consist of a mix of lowercase and uppercase letters, use the **compareToIgnoreCase** method instead

# - Building Boolean Expressions

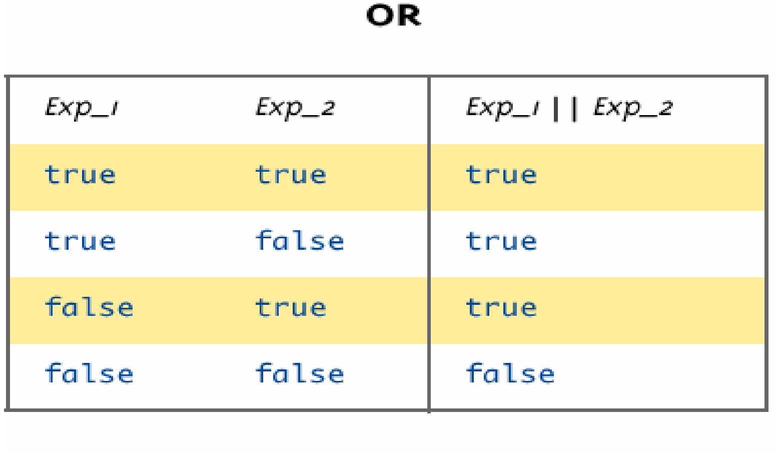
- When two Boolean expressions are combined using the *"and"* (**&&**) operator, the entire expression is true provided both expressions are true
  - Otherwise the expression is false
- When two Boolean expressions are combined using the *"or"* (**||**) operator, the entire expression is true as long as one of the expressions is true
  - The expression is false only if both expressions are false
- Any Boolean expression can be negated using the **!** Operator
  - Place the expression in parentheses and place the **!** operator in front of it
- Unlike mathematical notation, strings of inequalities must be joined by **&&**
  - Use **`(min < result) && (result < max)`** rather than **`min < result < max`**

# - Truth Tables



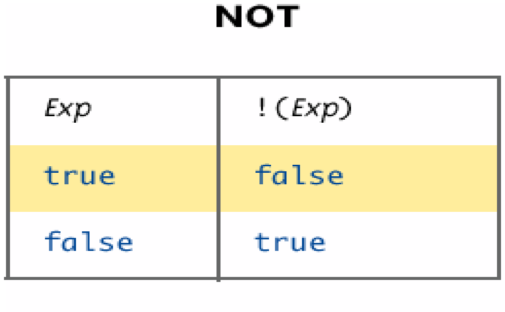
AND

| Exp_1 | Exp_2 | Exp_1 && Exp_2 |
|---|---|---|
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |



OR

| Exp_1 | Exp_2 | Exp_1 \|\| Exp_2 |
|---|---|---|
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |



NOT

| Exp | !(Exp) |
|---|---|
| true | false |
| false | true |

# - Short-Circuit and Complete Evaluation ...

- Consider `x > y || x > z`

- The expression is evaluated left to right. If `x > y` is `true`, then there's no need to evaluate `x > z` because the whole expression will be `true` whether `x > z` is `true` or not.

- This also happens when we use `&&` operator and the `first` expression is false.

- To stop the evaluation once the result of the whole expression is known is called *short-circuit evaluation* or *lazy evaluation*

# … - Short-Circuit and Complete Evaluation

- What would happen if the short-circuit evaluation is not done for the following expression?
  - **kids != 0 && toys/kids >= 2**
  - There are times when using short-circuit evaluation can prevent a *runtime error*
- Sometimes it is preferable to always evaluate both expressions, i.e., request complete evaluation
  - In this case, use the **&** and **|** operators instead of **&&** and **||**

# - Precedence and Associativity Rules ...

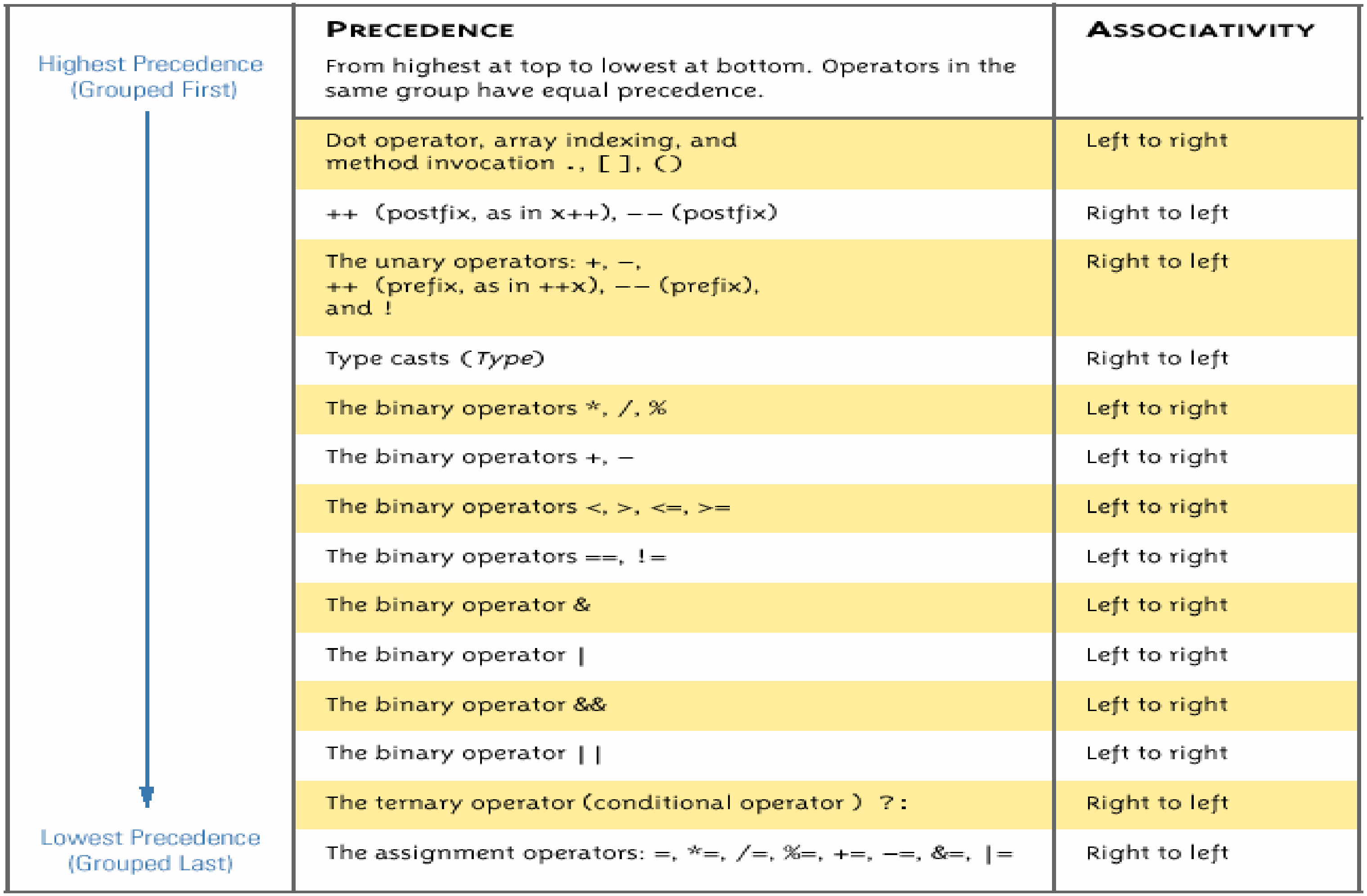
- Boolean and arithmetic expressions need not be fully parenthesized

- If some or all of the parentheses are omitted, Java will follow *precedence* and *associativity* rules (summarized in the following table) to determine the order of operations

  - If one operator occurs higher in the table than another, it has *higher precedence*, and is grouped with its operands before the operator of lower precedence

  - If two operators have the same precedence, then *associativity rules* determine which is grouped first

# … - Precedence and Associativity Rules



| Highest Precedence (Grouped First) | **PRECEDENCE**<br>From highest at top to lowest at bottom. Operators in the same group have equal precedence. | **ASSOCIATIVITY** |
|---|---|---|
| | Dot operator, array indexing, and method invocation ., [ ], ( ) | Left to right |
| | ++ (postfix, as in x++), −− (postfix) | Right to left |
| | The unary operators: +, −, ++ (prefix, as in ++x), −− (prefix), and ! | Right to left |
| | Type casts (*Type*) | Right to left |
| | The binary operators *, /, % | Left to right |
| | The binary operators +, − | Left to right |
| | The binary operators <, >, <=, >= | Left to right |
| | The binary operators ==, != | Left to right |
| | The binary operator & | Left to right |
| | The binary operator \| | Left to right |
| | The binary operator && | Left to right |
| | The binary operator \|\| | Left to right |
| | The ternary operator (conditional operator ) ?: | Right to left |
| Lowest Precedence (Grouped Last) | The assignment operators: =, *=, /=, %=, +=, −=, &=, \|= | Right to left |

# - Evaluating Expressions

- In general, parentheses in an expression help to document the programmer's intent
  - Instead of relying on precedence and associativity rules, it is best to include most parentheses, except where the intended meaning is obvious
- *Binding*: The association of operands with their operators
  - A fully parenthesized expression accomplishes binding for all the operators in an expression
- *Side Effects*: When, in addition to returning a value, an expression changes something, such as the value of a variable
  - The *assignment*, *increment*, and *decrement* operators all produce side effects

# - Rules for Evaluating Expressions

- Perform binding
    - Determine the equivalent fully parenthesized expression using the precedence and associativity rules
- Proceeding left to right, evaluate whatever subexpressions can be immediately evaluated
    - These subexpressions will be operands or method arguments, e.g., numeric constants or variables
- Evaluate each outer operation and method invocation as soon as all of its operands (i.e., arguments) have been evaluated

# THE END