Keenan Knaur
Adjunct Lecturer

California State University, Los Angeles
Computer Science Department

Selection Statements II

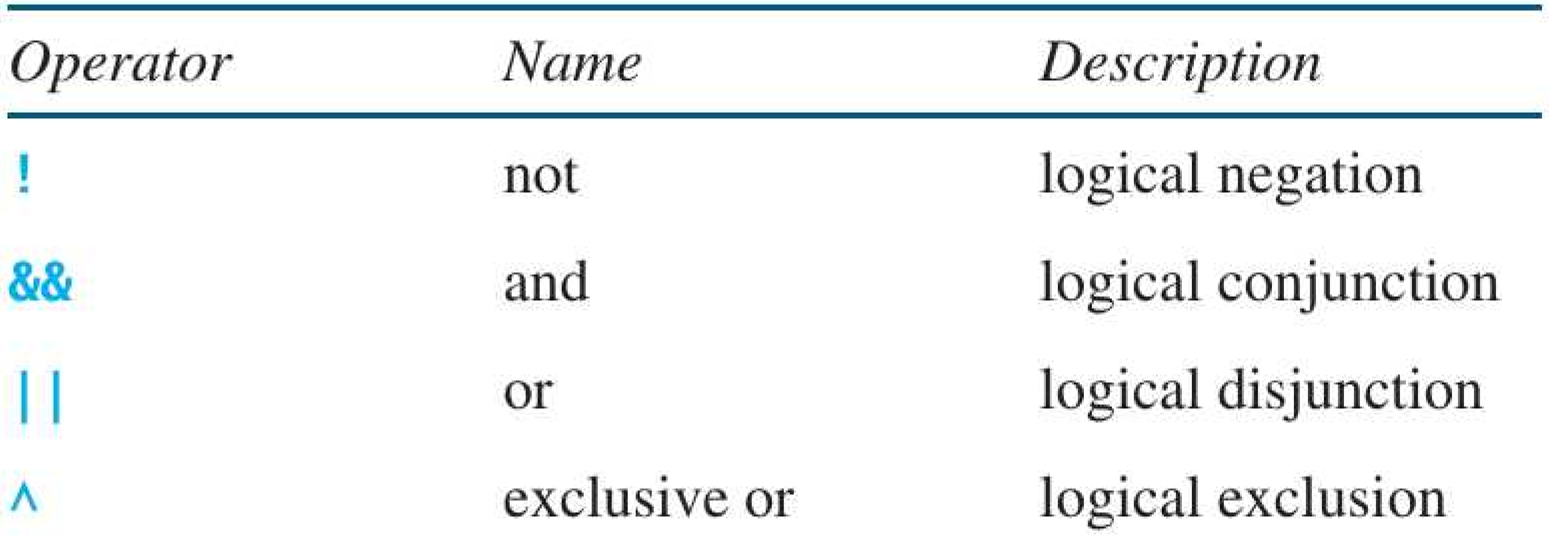
CS2011: Introduction to Programming I

Logical Operators

Logical Operators

- The logical operators `!` (not), `&&` (AND), `||` (OR), and `^` (exclusive OR) can be used to combine multiple conditions to form a compound Boolean expression.

- These ***logical operators***, also known as ***Boolean operators***, operate on Boolean values to create a new Boolean value.



Operator	*Name*	*Description*
!	not	logical negation
&&	and	logical conjunction
||	or	logical disjunction
^	exclusive or	logical exclusion

The ! (NOT) Operator

- Reverses the boolean value, **`true`** becomes **`false`** and **`false`** becomes **`true`**.

e	**!e**
true	false
false	true

The && (AND Operator)

- Evaluates to **true** if and only if both of the Boolean operands is **true.**
  - If one of the operands is false the entire expression is false.
- ***short-curcuit behavior***
  - When evaluating p1 && p2 Java first evaluates p1 and only will evaluate p2 if p1 is **true**.
  - If p1 is **false** it will not evaluate p2. This helps improve the performance of java.

e_1	e_2	e_1 && e_2
true	true	true
false	false	false
true	false	false
false	true	false

The || (OR) Operator

- Evaluates to true if at least one of the Boolean operands is **true**.
  - If one of the operands is true the entire expression is true.
- ***short-circuit behavior***:
  - When evaluating p1 || p2 Java first evaluates p1 and only will evaluate p2 if p1 is **false**.
  - If p1 is **true** it will not evaluate p2. This helps improve the performance of java.

e_1	e_2	$e_1 \| \| e_2$
true	true	true
false	false	false
true	false	true
false	true	true

The ^ (Exclusive OR) Operator

- Evaluates to **`true`** if and only if the two operands have OPPOSITE Boolean values.

e_1	e_2	e_1 || e_2
true	true	false
false	false	false
true	false	true
false	true	true

`switch` Statements

`switch` Statements

- A **switch** statement executes statements based on the value of a variable or an expression.
- Can be used to replace if and if-else statements when there are many alternatives.
- Can simplify coding for multiple conditions.

switch Statements

- Syntax:

```
switch (switch-expression) {
  case value1: statement(s)1;
               break;
  case value2: statement(s)2;
               break;
  ...
  case valueN: statement(s)N;
               break;
  default:     statement(s)-for-default;
}
```

switch Statements

- The switch-expression must yield a value of char, byte, short, int, or String type and must always be enclosed in parentheses.

```
switch (switch-expression) {
    case value1:   statement(s)1;
                   break;
    case value2:   statement(s)2;
                   break;
    case valueN:   statement(s)N;
                   break;
    default:       statement(s);
}
```

switch Statements

- The value1, ..., and valueN must have the same data type as the value of the switch-expression.
- The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression.
- Note that value1, ..., and valueN are constant expressions, meaning that they cannot contain variables in the expression, such as 1 + x.

```
switch (switch-expression) {

   case value1:   statement(s)1;

                  break;

   case value2:   statement(s)2;

                  break;

   case valueN:   statement(s)N;

                  break;

   default:       statement(s);

}
```

switch Statements

- The keyword break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement.

- If the break statement is not present, the next case statement will be executed

```
switch (switch-expression) {

   case value1:  statement(s)1;

                 break;

   case value2:  statement(s)2;

                 break;

   case valueN:  statement(s)N;

                 break;

   default:      statement(s);

}
```

switch Statements

- The default case, which is optional, can be used to perform actions when none of the specified cases matches the switch-expression.
- The case statements are executed in sequential order, but the order of the cases (including the default case) does not matter. However, it is good programming style to follow the logical sequence of the cases and place the default case at the end.

```
switch (switch-expression) {
    case value1:  statement(s)1;
                  break;
    case value2:  statement(s)2;
                  break;
    case valueN:  statement(s)N;
                  break;
    default:      statement(s);
}
```

switch Statements

- Don't forget to use a **break** statement when one is needed. Once a case is matched the statements starting from the matched case are executed until a **break** statement or the end of the **switch** statement. This is called the *fall-through behavior.*
- Example:

```
switch (ch) {
  case 'a': System.out.println(ch);
  case 'b': System.out.println(ch);
  case 'c': System.out.println(ch);
}
```

- Code Example: **`ChineseZodiac.java`**

if-else vs. Statements

```
if (score >= 90)
   grade = 'A';
else if (score >= 80)
   grade = 'B';
else if (score >= 70)
   grade = 'C';
else if (score >= 60)
   grade = 'D';
else
   grade = 'F';
```

```
switch (score / 10) {
  case 10:
  case 9:  grade = 'A';
           break;
  case 8:  grade = 'B';
           break;
  case 7:  grade = 'C';
           break;
  case 6:  grade = 'D';
           break;
  default: grade = 'F';
}
```

Conditional Expressions

Conditional Expressions

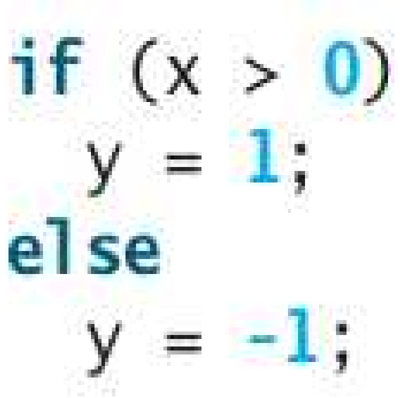
- A **conditional expression** evaluates an expressions based on a condition.
- A conditional expression can be used to replace a simple **if-else** statement.
- Syntax:

```
boolean-expression ? expression1 : expression2;
```

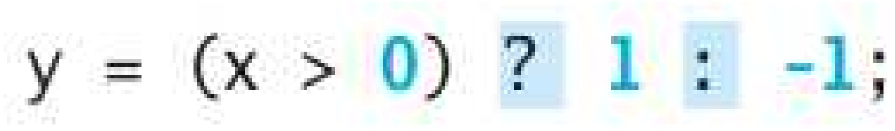
- The result of the conditional expression is expression1 if boolean-expression is **true**; otherwise the result is expression2

Conditional Expressions

- The first example assigns 1 to y if x is greater than 0, and -1 to y if x is less than or equal to 0.
- The second example is equivalent to the first and does the same thing using a conditional expression



```
if (x > 0)
   y = 1;
else
   y = -1;
```



```
y = (x > 0) ? 1 : -1;
```

- The symbols ? and : together form a conditional operator called a ***ternary operator*** because it uses three operands. It is the ONLY ternary operator in Java.

Conditional Expressions

- Example: Suppose you want to assign the larger number of variable **num1** and **num2** to **max**. You can simply write the following:

```
max = (num1 > num2) ? num1 : num2;
```

- Example: The following statement displays the message "num is even" if **num** is even and otherwise will display "num is odd"

```
System.out.println((num % 2 == 0) ?
  "num is even" : "num is odd");
```

Operator Precedence and Associativity

Operator Precedence and Associativity

- ***Operator precedence*** and ***associativity*** determine the order in which operators are evaluated.
  - parenthesis always evaluated first
  - nested parenthesis evaluated before outer parenthesis
  - operators are evaluated according to the precedence rule and the associativity rule.
  - if two operators with the same precedence are evaluated, the ***associativity*** of the operators determines the order of evaluation.
- All binary operators except assignment operators are ***left-associative*****.**
  - Example: a – b + c – d is equivalent to ((a – b) + c) – d
- The assignment operators are ***right-associative.***
  - Example: a = b += c = 5 is equivalent to a = (b += (c = 5))

Operator Precedence and Associativity

Precedence	*Operator*
	`var++` and `var--` (Postfix)
	`+`, `-` (Unary plus and minus), `++var` and `--var` (Prefix)
	(type) (Casting)
	`!` (Not)
	`*`, `/`, `%` (Multiplication, division, and remainder)
	`+`, `-` (Binary addition and subtraction)
	`<`, `<=`, `>`, `>=` (Relational)
	`==`, `!=` (Equality)
	`^` (Exclusive OR)
	`&&` (AND)
	`||` (OR)
	`=`, `+=`, `-=`, `*=`, `/=`, `%=` (Assignment operator)

References

- Liang, Chapter 03: Selection Statements