

Selections



EECS2030 E&F: Advanced
Object Oriented Programming
Fall 2024

CHEN-WEI WANG

Learning Outcomes

- The Boolean Data Type
- `if` Statement
- Compound vs. Primitive Statement
- Logical Operations
- Common Errors and Pitfalls

Extra Practice?

- Java project archive: `Lecture_02_Selections.zip` contains some of the lecture examples.
Expanded it: *remaining examples* and *your own examples* !
- *Optional (but recommended)*: **Videos 10 – 17** from W19 Java tutorial:
https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch

Motivating Examples (1.1)

```
1 import java.util.Scanner;
2 public class ComputeArea {
3     public static void main(String[] args) {
4         Scanner input = new Scanner(System.in);
5         System.out.println("Enter the radius of a circle:");
6         double radiusFromUser = input.nextDouble();
7         final double PI = 3.14;
8         double area = radiusFromUser * radiusFromUser * PI;
9         System.out.print("Circle with radius " + radiusFromUser);
10        System.out.println(" has an area of " + area);
11        input.close();
12    }
13 }
```

- When the above Java class is run as a Java Application, **Line 4** is executed first, followed by executing **Line 5**, ..., and ended with executing **Line 11**.
- In **Line 6**, the radius value comes from the user. Any problems?

Motivating Examples (1.2)

- If the user enters a positive radius value as expected:

```
Enter the radius of a circle:
```

```
3
```

```
Circle with radius 3.0 has an area of 28.26
```

- However, if the user enters a negative radius value:

```
Enter the radius of a circle:
```

```
-3
```

```
Circle with radius -3.0 has an area of 28.26
```

In this case, the area should *not* have been calculated!

- We need a mechanism to take *selective actions*:

Act differently in response to *valid* and *invalid* input values.

Motivating Examples (2.1)

Problem: Take an integer value from the user, then output a message indicating if the number is negative, zero, or positive.

- Here is an example run of the program:

```
Enter a number:
```

```
5
```

```
You just entered a positive number.
```

- Here is another example run of the program:

```
Enter a number:
```

```
-5
```

```
You just entered a negative number.
```

- Your solution program must accommodate *all* possibilities!

Motivating Examples (2.2)

- So far, you only learned about writing programs that are executed top to bottom, line by line, without ever **branching**.
- In general, we need a mechanism to allow the program to:
 - Check a list of **conditions**; and
 - **Branch** its execution accordingly.
- e.g., To solve the above problem, we have 3 possible branches:
 1. **If** the user input is negative, then we execute the first branch that prints `You just entered a negative number.`
 2. **If** the user input is zero, then we execute the second branch that prints `You just entered zero.`
 3. **If** the user input is positive, then we execute the third branch that prints `You just entered a positive number.`

The boolean Data Type

- A (data) type denotes a set of related *runtime values*.
- We need a **data type** whose values suggest either a condition *holds*, or it *does not hold*, so that we can take selective actions.
- The Java **boolean** type consists of 2 **literal values**: *true*, *false*
- All **relational expressions** have the boolean type.

Math Symbol	Java Operator	Example (<i>r</i> is 5)	Result
\leq	<code><=</code>	<code>r <= 5</code>	<i>true</i>
\geq	<code>>=</code>	<code>r >= 5</code>	<i>true</i>
$=$	<code>==</code>	<code>r == 5</code>	<i>true</i>
$<$	<code><</code>	<code>r < 5</code>	<i>false</i>
$>$	<code>></code>	<code>r > 5</code>	<i>false</i>
\neq	<code>!=</code>	<code>r != 5</code>	<i>false</i>

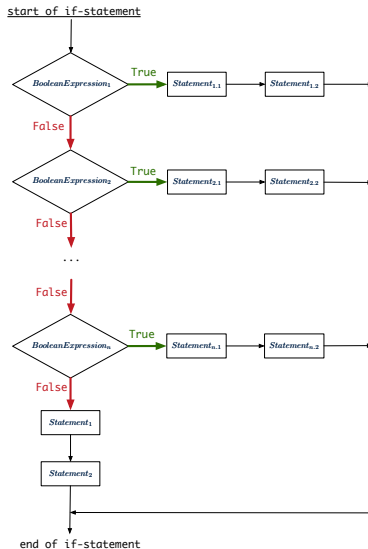
Note. You may do the following rewritings:

$x \leq y$	$x > y$	$x \neq y$	$x == y$
$!(x > y)$	$!(x \leq y)$	$!(x == y)$	$!(x \neq y)$

Syntax of if Statement

```
if ( BooleanExpression1 ) { /* Mandatory */  
    Statement1.1; Statement2.1;  
}  
else if ( BooleanExpression2 ) { /* Optional */  
    Statement2.1; Statement2.2;  
}  
... /* as many else-if branches as you like */  
else if ( BooleanExpressionn ) { /* Optional */  
    Statementn.1; Statementn.2;  
}  
else { /* Optional */  
    /* when all previous branching conditions are false */  
    Statement1; Statement2;  
}
```

Semantics of if Statement (1.1)



Semantics of `if` Statement (1.2)

Consider a *single if statement* as consisting of:

- An `if` branch
- A (possibly empty) list of `else if` branches
- An optional `else` branch

At *runtime*:

- Branches of the `if` statement are *executed* from top to bottom.
- We only evaluate the **condition** of a branch if those conditions of its **preceding branches** evaluate to *false*.
- The **first** branch whose **condition** evaluates to *true* gets its body (i.e., code wrapped within { and }) *executed*.
 - After this execution, all *later* branches are *ignored*.

Semantics of `if` Statement: Case 1

Only first satisfying branch *executed*; later branches *ignored*.

```
int i = -4;
if(i < 0) {
    System.out.println("i is negative");
}
else if(i < 10) {
    System.out.println("i is less than than 10");
}
else if(i == 10) {
    System.out.println("i is equal to 10");
}
else {
    System.out.println("i is greater than 10");
}
```

i is negative

Semantics of `if` Statement: Case 2

Only first satisfying branch *executed*; later branches *ignored*.

```
int i = 5;
if(i < 0) {
    System.out.println("i is negative");
}
else if(i < 10) {
    System.out.println("i is less than than 10");
}
else if(i == 10) {
    System.out.println("i is equal to 10");
}
else {
    System.out.println("i is greater than 10");
}
```

i is less than 10

Semantics of `if` Statement: Case 3

Only first satisfying branch *executed*; later branches *ignored*.

```
int i = 10;  
if(i < 0) {  
    System.out.println("i is negative");  
}  
else if(i < 10) {  
    System.out.println("i is less than than 10");  
}  
else if(i == 10) {  
    System.out.println("i is equal to 10");  
}  
else {  
    System.out.println("i is greater than 10");  
}
```

i is equal to 10

Semantics of `if` Statement: Case 4

No satisfying branches, and an `else` part is present, then the *default action* is executed.

```
int i = 12;  
if(i < 0) {  
    System.out.println("i is negative");  
}  
else if(i < 10) {  
    System.out.println("i is less than than 10");  
}  
else if(i == 10) {  
    System.out.println("i is equal to 10");  
}  
else {  
    System.out.println("i is greater than 10");  
}
```

i is greater than 10

Semantics of `if` Statement: Case 5

No satisfying branches, and an `else` part is absent, then *nothing* is executed.

```
int i = 12;  
if(i < 0) {  
    System.out.println("i is negative");  
}  
else if(i < 10) {  
    System.out.println("i is less than than 10");  
}  
else if(i == 10) {  
    System.out.println("i is equal to 10");  
}
```


Logical Operators

- *Logical* operators are used to create **compound** Boolean expressions.
 - Similar to *arithmetic* operators for creating compound number expressions.
 - *Logical* operators can combine Boolean expressions that are built using the *relational* operators.

e.g., `1 <= x && x <= 10`

e.g., `x < 1 || x > 10`

- We consider three logical operators:

Java Operator	Description	Meaning
<code>!</code>	logical negation	not
<code>&&</code>	logical conjunction	and
<code> </code>	logical disjunction	or

Logical Negation

- Logical **negation** is a *unary* operator (i.e., one operand being a Boolean expression).
- The result is the “negated” value of its operand.

Operand	op	!op
<i>true</i>		<i>false</i>
<i>false</i>		<i>true</i>

```
double radius = input.nextDouble();
final double PI = 3.14;
boolean isPositive = radius > 0;
if (!isPositive) { /* not the case that isPositive is true */
    System.out.println("Error: radius value must be positive.");
}
else {
    System.out.println("Area is " + radius * radius * PI);
}
```

Logical Conjunction

- Logical **conjunction** is a *binary* operator (i.e., two operands, each being a Boolean expression).
- The conjunction is *true* only when both operands are *true*.
- If one of the operands is *false*, their conjunction is *false*.

Left Operand op1	Right Operand op2	op1 && op2
<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

```

int age = input.nextInt();
boolean isOldEnough = age >= 45;
boolean isNotTooOld = age < 65;
if (!isOldEnough) { /* young */ }
else if (isOldEnough && isNotTooOld) { /* middle-aged */ }
else { /* senior */ }

```

Logical Disjunction

- Logical **disjunction** is a *binary* operator (i.e., two operands, each being a Boolean expression).
- The disjunction is *false* only when both operands are *false*.
- If one of the operands is *true*, their disjunction is *true*.

Left Operand op1	Right Operand op2	op1 op2
<i>false</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>true</i>	<i>true</i>

```

int age = input.nextInt();
boolean isSenior = age >= 65;
boolean isChild = age < 18;
if (isSenior || isChild) { /* discount */ }
else { /* no discount */ }

```

Logical Laws: Negation

- The **negation** of a strict inequality is a non-strict inequality.

Relation	Negation	Equivalence
$i > j$	$!(i > j)$	$i \leq j$
$i \geq j$	$!(i \geq j)$	$i < j$
$i < j$	$!(i < j)$	$i \geq j$
$i \leq j$	$!(i \leq j)$	$i > j$

- e.g.,

```

if(  $i > j$  ) {
    /* Action 1 */
}
else { /*  $!(i > j)$  */
    /* Action 2 */
}

```

equivalent to

```

if(  $i \leq j$  ) {
    /* Action 2 */
}
else { /*  $!(i \leq j)$  */
    /* Action 1 */
}

```

- Action 1 is executed when $i > j$
- Action 2 is executed when $i \leq j$.

Case Study: Error Handling of Input Radius (1)

Problem: Prompt user for radius value of a circle. Print an error message if input is negative; otherwise, print the calculated area.

```
public class ComputeArea {  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        System.out.println("Enter a radius value:");  
        double radius = input.nextDouble();  
        final double PI = 3.14159;  
        if (radius < 0) { /* condition of invalid inputs */  
            System.out.println("Error: Negative radius value!");  
        }  
        else { /* implicit: !(radius < 0), or radius >= 0 */  
            double area = radius * radius * PI;  
            System.out.println("Area is " + area);  
        }  
        input.close();  
    }  
}
```

Case Study: Error Handling of Input Radius (2)



The same problem can be solved by checking the *condition* of valid inputs first.

```
public class ComputeArea2 {  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        System.out.println("Enter a radius value:");  
        double radius = input.nextDouble();  
        final double PI = 3.14159;  
        if (radius >= 0) { /* condition of valid inputs */  
            double area = radius * radius * PI;  
            System.out.println("Area is " + area);  
        }  
        else { /* implicit: !(radius >= 0), or radius < 0 */  
            System.out.println("Error: Negative radius value!");  
        }  
        input.close();  
    }  
}
```

Logical Laws: DeMorgan for Conjunction

Say we have two Boolean expressions B_1 and B_2 :

- What does $\neg (B_1 \ \&\& \ B_2)$ mean?

It is **not** the case that both B_1 and B_2 are *true*.

- What does $\neg B_1 \ || \ \neg B_2$ mean?

It is either B_1 is *false*, B_2 is *false*, or both are *false*.

- Both expressions are equivalent! [proved by the truth table]

B_1	B_2	$\neg (B_1 \ \&\& \ B_2)$	$\neg B_1 \ \ \neg B_2$
<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>

Logical Laws: DeMorgan for Disjunction

Say we have two Boolean expressions B_1 and B_2 :

- What does $!(B_1 \vee B_2)$ mean?

It is **not** the case that either B_1 is *true*, B_2 is *true*, or both are *true*.

- What does $!B_1 \&\&!B_2$ mean?

Both B_1 and B_2 are *false*.

- Both expressions are equivalent! [proved by the truth table]

B_1	B_2	$!(B_1 \vee B_2)$	$!B_1 \&\&!B_2$
<i>true</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>

Example: DeMorgan for Conjunction

```
if(0 <= i && i <= 10) { /* Action 1 */ }  
else { /* Action 2 */ }
```

- When is *Action 2* executed?

$i < 0 \mid\mid i > 10$

```
if(i < 0 && false) { /* Action 1 */ }  
else { /* Action 2 */ }
```

- When is *Action 1* executed? *false*
- When is *Action 2* executed? *true* (i.e., $i \geq 0 \mid\mid \text{true}$)

```
if(i < 0 && i > 10) { /* Action 1 */ }  
else { /* Action 2 */ }
```

- When is *Action 1* executed? *false*
- When is *Action 2* executed? *true* (i.e., $i \geq 0 \mid\mid i \leq 10$)

Lesson: Be careful not to write branching conditions that use `&&` but always evaluate to *false*.

Example: DeMorgan for Disjunction

```
if(i < 0 || i > 10) { /* Action 1 */ }  
else { /* Action 2 */ }
```

- When is *Action 2* executed? $0 \leq i \ \&\& \ i \leq 10$

```
if(i < 0 || true) { /* Action 1 */ }  
else { /* Action 2 */ }
```

- When is *Action 1* executed? *true*
- When is *Action 2* executed? *false* (i.e., $i \geq 0 \ \&\& \ \text{false}$)

```
if(i < 10 || i >= 10) { /* Action 1 */ }  
else { /* Action 2 */ }
```

- When is *Action 1* executed? *true*
- When is *Action 2* executed? *false* (i.e., $i \geq 10 \ \&\& \ i < 10$)

Lesson: Be careful not to write branching conditions that use `||` but always evaluate to *true*.

Operator Precedence

- Operators with *higher* precedence are evaluated before those with *lower* precedence.

e.g., $2 + 3 * 5$

- For the three *logical operators*, negation (!) has the highest precedence, then conjunction (&&), then disjunction (||).

e.g., `true || true && false` means

- `true || (true && false)`, rather than
- `(true || true) && false`

- When unsure, use *parentheses* to force the precedence.

Operator Associativity

- When operators with the *same precedence* are grouped together, we evaluate them from left to right.

e.g., $1 + 2 - 3$ means

$((1 + 2) - 3)$

e.g., `false || true || false` means

`((false || true) || false)`

Two-Way `if` Statement without `else` Part

```
if (radius >= 0) {  
    area = radius * radius * PI;  
    System.out.println("Area for the circle of is " + area);  
}
```

An `if` statement with the missing `else` part is equivalent to an `if` statement with an `else` part that does nothing.

```
if (radius >= 0) {  
    area = radius * radius * PI;  
    System.out.println("Area for the circle of is " + area);  
}  
else {  
    /* Do nothing. */  
}
```

Primitive Statement vs. Compound Statement

- A **statement** is a block of Java code that modifies value(s) of some variable(s).
- An assignment (=) statement is a *primitive statement*:
It only modifies its left-hand-side (LHS) variable.
- An `if` statement is a *compound statement*:
Each of its branches may modify more than one variables via other statements (e.g., assignments, `if` statements).

Compound if Statement: Example

```
1  int x = input.nextInt();
2  int y = 0;
3  if (x >= 0) {
4      System.out.println("x is positive");
5      if (x > 10) { y = x * 2; }
6      else if (x < 10) { y = x % 2; }
7      else { y = x * x; }
8  }
9  else { /* x < 0 */
10     System.out.println("x is negative");
11     if(x < -5) { y = -x; }
12 }
```

Exercise: Draw a flow chart for the above compound statement.

Multi-Way if Statement with else Part

```
if (score >= 80.0) {  
    System.out.println("A");  
}  
else if (score >= 70.0) {  
    System.out.println("B");  
}  
else if (score >= 60.0) {  
    System.out.println("C");  
}  
else {  
    System.out.println("F");  
}
```

```
if (score >= 80.0) {  
    System.out.println("A"); }  
else { /* score < 80.0 */  
    if (score >= 70.0) {  
        System.out.println("B"); }  
    else { /* score < 70.0 */  
        if (score >= 60.0) {  
            System.out.println("C"); }  
        else { /* score < 60.0 */  
            System.out.println("F");  
        }  
    }  
}
```

Exercise: Draw the corresponding flow charts for both programs. Convince yourself that they are equivalent.

Multi-Way `if` Statement without `else` Part

```
String letterGrade = "F";  
if (score >= 80.0) {  
    letterGrade = "A";  
}  
else if (score >= 70.0) {  
    letterGrade = "B";  
}  
else if (score >= 60.0) {  
    letterGrade = "C";  
}
```

In this case, since we already assign an initial, default value "F" to variable `letterGrade`, so when all the branch conditions evaluate to *false*, then the default value is kept.

Compare the above example with the example in slide 56.

One if Stmt vs. Multiple if Stmts (1)

Question: Do these two programs behave same at runtime?

```
if(i >= 3) {System.out.println("i is >= 3");}  
else if(i <= 8) {System.out.println("i is <= 8");}
```

```
if(i >= 3) {System.out.println("i is >= 3");}  
if(i <= 8) {System.out.println("i is <= 8");}
```

Question: Do these two programs behave same at runtime?

```
if(i <= 3) {System.out.println("i is <= 3");}  
else if(i >= 8) {System.out.println("i is >= 8");}
```

```
if(i <= 3) {System.out.println("i is <= 3");}  
if(i >= 8) {System.out.println("i is >= 8");}
```

One if Stmt vs. Multiple if Stmts (2)

```
int i = 5;  
if(i >= 3) {System.out.println("i is >= 3");}  
else if(i <= 8) {System.out.println("i is <= 8");}
```

i is >= 3

```
int i = 5;  
if(i >= 3) {System.out.println("i is >= 3");}  
if(i <= 8) {System.out.println("i is <= 8");}
```

i is >= 3
i is <= 8

Two versions behave *differently* because the two conditions $i \geq 3$ and $i \leq 8$ *may* be satisfied simultaneously.

One if Stmt vs. Multiple if Stmts (3)

```
int i = 2;  
if(i <= 3) {System.out.println("i is <= 3");}  
else if(i >= 8) {System.out.println("i is >= 8");}
```

```
i is <= 3
```

```
int i = 2;  
if(i <= 3) {System.out.println("i is <= 3");}  
if(i >= 8) {System.out.println("i is >= 8");}
```

```
i is <= 3
```

Two versions behave *the same* because the two conditions $i \leq 3$ and $i \geq 8$ *cannot* be satisfied simultaneously.

Common Error 1: Independent `if` Statements with Overlapping Conditions

```
if (marks >= 80) {  
    System.out.println("A");  
}  
if (marks >= 70) {  
    System.out.println("B");  
}  
if (marks >= 60) {  
    System.out.println("C");  
}  
else {  
    System.out.println("F");  
}  
/* Consider marks = 84 */
```

```
if (marks >= 80) {  
    System.out.println("A");  
}  
else if (marks >= 70) {  
    System.out.println("B");  
}  
else if (marks >= 60) {  
    System.out.println("C");  
}  
else {  
    System.out.println("F");  
}  
/* Consider marks = 84 */
```

- *Conditions* in a list of `if` statements are checked **independently**.
- In a single `if` statement, **only** the *first satisfying branch* is executed.

Overlapping Conditions: Exercise (1)

- Does this program always print exactly one line?

```
if(x < 0) { println("x < 0"); }  
if(0 <= x && x < 10) { println("0 <= x < 10"); }  
if(10 <= x && x < 20) { println("10 <= x < 20"); }  
if(x >= 20) { println("x >= 20"); }
```

- Yes**, because the branching conditions for the **four** if-statements are all **non-overlapping**.
- That is, any two of these conditions **cannot be satisfied simultaneously**:
 - $x < 0$
 - $0 \leq x \text{ \&\& } x < 10$
 - $10 \leq x \text{ \&\& } x < 20$
 - $x \geq 20$

Overlapping Conditions: Exercise (2)

- Does this program always print exactly one line?

```
if(x < 0) { println("x < 0"); }  
else if(0 <= x && x < 10) { println("0 <= x < 10"); }  
else if(10 <= x && x < 20) { println("10 <= x < 20"); }  
else if(x >= 20) { println("x >= 20"); }
```

- Yes**, because it's a **single** if-statement:

Only **the first satisfying branch** is executed.

- But, can it be simplified?

Hint: In a single if-statement, a branch is executed only if **all earlier branching conditions** fail.

Overlapping Conditions: Exercise (3)

- This simplified version is equivalent:

```
1  if(x < 0) { println("x < 0"); }  
2  else if(x < 10) { println("0 <= x < 10"); }  
3  else if(x < 20) { println("10 <= x < 20"); }  
4  else { println("x >= 20"); }
```

- At runtime, the 2nd condition `x < 10` at **L2** is checked only when the 1st condition at **L1** *fails* (i.e., $!(x < 0)$, or equivalently, $x \geq 0$).
- At runtime, the 3rd condition `x < 20` at **L3** is checked only when the 2nd condition at **L2** *fails* (i.e., $!(x < 10)$, or equivalently, $x \geq 10$).
- At runtime, the else (default) branch at **L4** is reached only when the 3rd condition at **L3** *fails* (i.e., $!(x < 20)$, or equivalently, $x \geq 20$).

Scope of Variables (1)

When you declare a variable, there is a limited **scope** where the variable can be used.

- If the variable is declared directly under the `main` method, then all lines of code (including branches of `if` statements) may either **re-assign** a new value to it or **use** its value.

```
public static void main(String[] args) {  
    int i = input.nextInt();  
    System.out.println("i is " + i);  
    if (i > 0) {  
        i = i * 3; /* both use and re-assignment, why? */  
    }  
    else {  
        i = i * -3; /* both use and re-assignment, why? */  
    }  
    System.out.println("3 * |i| is " + i);  
}
```

Scope of Variables (2.1)

- If the variable is declared under an `if` branch, an `else if` branch, or an `else` branch, then only lines of code appearing within that branch (i.e., its body) may either *re-assign* a new value to it or *use* its value.

```
public static void main(String[] args) {  
    int i = input.nextInt();  
    if (i > 0) {  
        int j = i * 3; /* a new variable j */  
        if (j > 10) { ... }  
    }  
    else {  
        int j = i * -3; /* a new variable also called j */  
        if (j < 10) { ... }  
    }  
}
```

Scope of Variables (2.2)

- A variable declared under an `if` branch, an `else if` branch, or an `else` branch, cannot be *re-assigned* or *used* outside its scope.

```
public static void main(String[] args) {  
    int i = input.nextInt();  
    if (i > 0) {  
        int j = i * 3; /* a new variable j */  
        if (j > 10) { ... }  
    }  
    else {  
        int k = i * -3; /* a new variable also called j */  
        if (j < k) { ... }    x  
    }  
}
```

Scope of Variables (2.3)

- A variable declared under an `if` branch, `else if` branch, or `else` branch, cannot be *re-assigned* or *used* outside its scope.

```

1 public static void main(String[] args) {
2     int i = input.nextInt();
3     if (i > 0) {
4         int j = i * 3; /* a new variable j */
5         if (j > 10) { ... }
6     }
7     else {
8         int j = i * -3; /* a new variable also called j */
9         if (j < 10) { ... }
10    }
11    System.out.println("i * j is " + (i * j));    ×
12 }

```

- A variable *cannot* be referred to outside its declared scope.
[e.g., illegal use of `j` at **L11**]
- A variable *can* be used:
 - within its declared scope [e.g., use of `i` at **L11**]
 - within sub-scopes of its declared scope [e.g., use of `i` at **L4**, **L8**]

Scope of Variables (2.4)

How about **input** parameters and **return** value?

```

1 public class SumApp {
2     public static void main(String[] args) {
3         Scanner input = new Scanner(System.in);
4         int i = input.nextInt();
5         int j = input.nextInt();
6         int k = Utilities.getSum(i, j);
7         System.out.println(k);
8     } }

```

```

public class Utilities {
    public static int getSum(int x, int y) {
        int result = x + y;
        return result;
    } }

```

- Scope of i, j, k? [SumApp.main]
- Scope of x, y, result? [Utilities.getSum]
- **L5** is as if we wrote: `int k = result;`
 where `result` stores the value computed by executing `getSum`

General vs. Specific Boolean Conditions (1)

Two or more conditions **overlap** if they can evaluate to *true* simultaneously.

e.g., Say `marks` is declared as an integer variable:

- `marks >= 80` and `marks >= 70` overlap. [why?]
 - Values 80, 81, 82, ... make both conditions *true*
 - `marks >= 80` has **fewer** satisfying values than `marks >= 70`
 - We say `marks >= 80` is more *specific* than `marks >= 70`
 - Or, we say `marks >= 70` is more *general* than `marks >= 80`
- `marks <= 65` and `marks <= 75` overlap. [why?]
 - Values 65, 64, 63, ... make both conditions *true*
 - `marks <= 65` has **fewer** satisfying values than `marks <= 75`
 - We say `marks <= 65` is more *specific* than `marks <= 75`
 - Or, we say `marks <= 75` is more *general* than `marks <= 65`

General vs. Specific Boolean Conditions (2)

Say we have two overlapping conditions $x \geq 5$ and $x \geq 0$:

- What values make both conditions *true*? [5, 6, 7, ...]
- Which condition is more *general*? [$x \geq 0$]
- If we have a single if statement, then having this order

```
if(x >= 5) { System.out.println("x >= 5"); }  
else if(x >= 0) { System.out.println("x >= 0"); }
```

is different from having this order

```
if(x >= 0) { System.out.println("x >= 0"); }  
else if(x >= 5) { System.out.println("x >= 5"); }
```

- Say x is 5, then we have
 - What output from the first program? [$x \geq 5$]
 - What output from the second program? [$x \geq 0$, not *specific* enough!]
- The cause of the “*not-specific-enough*” problem of the second program is that we did not check the more *specific* condition ($x \geq 5$) before checking the more *general* condition ($x \geq 0$).

Common Error 2: `if-elseif` Statement with Most General Condition First (1)

```
if (gpa >= 2.5) {  
    graduateWith = "Pass";  
}  
else if (gpa >= 3.5) {  
    graduateWith = "Credit";  
}  
else if (gpa >= 4) {  
    graduateWith = "Distinction";  
}  
else if (gpa >= 4.5) {  
    graduateWith = "High Distinction" ;  
}
```

The above program will:

- Not award a “High Distinction” to `gpa == 4.8`.
- Why?

Common Error 2: if-elseif Statement with Most General Condition First (2)

- Always “*sort*” the branching conditions s.t. the more *specific* conditions are checked before the more *general* conditions.

```
if (gpa >= 4.5) {  
    graduateWith = "High Distinction" ;  
}  
else if (gpa >= 4) {  
    graduateWith = "Distinction";  
}  
else if (gpa >= 3.5) {  
    graduateWith = "Credit";  
}  
else if (gpa >= 2.5) {  
    graduateWith = "Pass";  
}  
else { graduateWith = "Fail"; }
```

Short-Circuit Evaluation (1)

- Both *Logical operators* `&&` and `||` evaluate from left to right.
- Operator `&&` continues to evaluate only when operands so far evaluate to *true*.

```
if (x != 0 && y / x > 2) {  
    /* do something */  
}  
else {  
    /* print error */ }
```

- Operator `||` continues to evaluate only when operands so far evaluate to *false*.

```
if (x == 0 || y / x <= 2) {  
    /* print error */  
}  
else {  
    /* do something */ }
```

Short-Circuit Evaluation (2)

- Both *Logical operators* `&&` and `||` evaluate from left to right.
- Short-Circuit Evaluation is not exploited: crash when `x == 0`

```
if (y / x > 2 && x != 0) {  
    /* do something */  
}  
else {  
    /* print error */ }
```

- Short-Circuit Evaluation is not exploited: crash when `x == 0`

```
if (y / x <= 2 || x == 0) {  
    /* print error */  
}  
else {  
    /* do something */ }
```

Common Error 3: Missing Braces (1)

Confusingly, braces can be omitted if the block contains a **single** statement.

```
final double PI = 3.1415926;  
Scanner input = new Scanner(System.in);  
double radius = input.nextDouble();  
if (radius >= 0)  
    System.out.println("Area is " + radius * radius * PI);
```

In the above code, it is as if we wrote:

```
final double PI = 3.1415926;  
Scanner input = new Scanner(System.in);  
double radius = input.nextDouble();  
if (radius >= 0) {  
    System.out.println("Area is " + radius * radius * PI);  
}
```

Common Error 3: Missing Braces (2)

Your program will *misbehave* when a block is supposed to execute **multiple statements**, but you forget to enclose them within braces.

```
final double PI = 3.1415926;  
Scanner input = new Scanner(System.in);  
double radius = input.nextDouble();  
double area = 0;  
if (radius >= 0)  
    area = radius * radius * PI;  
    System.out.println("Area is " + area);
```

This program will **mistakenly** print “Area is 0.0” when a *negative* number is input by the user, why? Fix?

```
if (radius >= 0) {  
    area = radius * radius * PI;  
    System.out.println("Area is " + area);  
}
```

Common Error 4: Misplaced Semicolon

Semicolon (;) in Java marks *the end of a statement* (e.g., assignment, if statement).

```
if (radius >= 0); {  
    area = radius * radius * PI;  
    System.out.println("Area is " + area);  
}
```

This program will calculate and output the area even when the input radius is *negative*, why? Fix?

```
if (radius >= 0) {  
    area = radius * radius * PI;  
    System.out.println("Area is " + area);  
}
```

Common Error 5: Variable Not Properly Re-Assigned

```
1 String graduateWith = "";
2 if (gpa >= 4.5) {
3     graduateWith = "High Distinction" ; }
4 else if (gpa >= 4) {
5     graduateWith = "Distinction"; }
6 else if (gpa >= 3.5) {
7     graduateWith = "Credit"; }
8 else if (gpa >= 2.5) {
9     graduateWith = "Pass"; }
```

The above program will award "" to $gpa == 1.5$. Why?

Possible Fix 1: Change the *initial value* in Line 1 to "Fail".

Possible Fix 2: Add an *else* branch after Line 9:

```
else { graduateWith = "fail" }
```

Compare this example with the example in slide 34.

Common Errors 6: Ambiguous else (1)

```
if (x >= 0)
    if (x > 100) {
        System.out.println("x is larger than 100");
    }
else {
    System.out.println("x is negative");
}
```

- When x is 20, this program considers it as negative. Why?
∴ else clause matches the *most recent* unmatched if clause.
∴ The above is as if we wrote:

```
if (x >= 0) {
    if (x > 100) {
        System.out.println("x is larger than 100");
    }
    else {
        System.out.println("x is negative");
    }
}
```

Common Errors 6: Ambiguous else (2)

- Fix?

Use pairs of curly braces ({}) to force what you really mean to specify!

```
if (x >= 0) {  
    if (x > 100) {  
        System.out.println("x is larger than 100");  
    }  
}  
else {  
    System.out.println("x is negative");  
}
```

Common Pitfall 1: Updating Boolean Variable

```
boolean isEven;  
if (number % 2 == 0) {  
    isEven = true;  
}  
else {  
    isEven = false;  
}
```

Correct, but **simplifiable**: `boolean isEven = (number%2 == 0);`
Similarly, how would you simplify the following?

```
if (isEven == false) {  
    System.out.println("Odd Number");  
}  
else {  
    System.out.println("Even Number");  
}
```

Simplify `isEven == false` to `!isEven`

Beyond this lecture...

- Create a *console tester* in Eclipse.
Try out the examples given in the slides.
- Solve the motivating example in **Slide 5**.
- *Optional (but recommended)*: **Videos 10 – 17** from W19 Java tutorial:
`https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch`

Index (1)

Learning Outcomes

Extra Practice?

Motivating Examples (1.1)

Motivating Examples (1.2)

Motivating Examples (2.1)

Motivating Examples (2.2)

The `boolean` Data Type

Syntax of `if` Statement

Semantics of `if` Statement (1.1)

Semantics of `if` Statement (1.2)

Semantics of `if` Statement: Case 1

Index (2)

Semantics of `if` Statement: Case 2

Semantics of `if` Statement: Case 3

Semantics of `if` Statement: Case 4

Semantics of `if` Statement: Case 5

Logical Operators

Logical Operators: Negation

Logical Operators: Conjunction

Logical Operators: Disjunction

Logical Laws: Negation

Case Study: Error Handling of Input Radius (1)

Case Study: Error Handling of Input Radius (2)

Index (3)

Logical Laws: DeMorgan for Conjunction

Logical Laws: DeMorgan for Disjunction

Example: DeMorgan for Conjunction

Example: DeMorgan for Disjunction

Operator Precedence

Operator Associativity

Two-Way `if` Statement without `else` Part

Primitive Statement vs. Compound Statement

Compound `if` Statement: Example

Multi-Way `if` Statement with `else` Part

Multi-Way `if` Statement without `else` Part

Index (4)

One `if` Stmt vs. Multiple `if` Stmts (1)

One `if` Stmt vs. Multiple `if` Stmts (2)

One `if` Stmt vs. Multiple `if` Stmts (3)

Common Error 1: Independent `if` Statements with
Overlapping Conditions

Overlapping Conditions: Exercise (1)

Overlapping Conditions: Exercise (2)

Overlapping Conditions: Exercise (3)

Scope of Variables (1)

Scope of Variables (2.1)

Scope of Variables (2.2)

Index (5)

Scope of Variables (2.3)

Scope of Variables (2.4)

General vs. Specific Boolean Conditions (1)

General vs. Specific Boolean Conditions (2)

Common Error 2: `if-elseif` Statement with Most General Condition First (1)

Common Error 2: `if-elseif` Statement with Most General Condition First (2)

Short-Circuit Evaluation (1)

Short-Circuit Evaluation (2)

Common Error 3: Missing Braces (1)

Index (6)

Common Error 3: Missing Braces (2)

Common Error 4: Misplaced Semicolon

Common Error 5:

Variable Not Properly Re-Assigned

Common Error 6: Ambiguous `else` (1)

Common Error 6: Ambiguous `else` (2)

Common Pitfall 1: Updating Boolean Variable

Beyond this lecture...