COMP-202
Unit 3: Conditional Programming

CONTENTS:
Boolean Expressions
The if and if-else Statements

Introduction

• Suppose we want to write a program which asks the user to enter two numbers and then displays only the larger of the two.
• This will involve executing certain statements in some circumstances, and different statements in other circumstances.
• Problem: So far, all the programs we have written executed all the statements they contained all the time.
  – We do not yet know the tools to make decisions about which statements should be executed.

Control Flow

• The default order of statement execution through a method is linear: one statement after the other, in the order they are written (from the top of the page down towards the bottom).
• Some programming statements modify that order, allowing us to:
  – decide whether or not to execute some statements, or
  – perform some statements over and over repetitively.
• The order of statement execution is called control flow or flow of control.

Aside: Fundamental Structures (1)

• A control flow structure is a basic unit of programming logic.
• Any program can be constructed using only three structures:
  – Sequence
  – Selection / decision / conditional
  – Repetition / iteration / loop
• The most common programming languages support these three structures.
Aside: Fundamental Structures (2)

- In the sequence structure, statements are executed in the order they appear in the code

- This is what we have seen so far

Aside: Fundamental Structures (3)

- In the selection / decision / conditional structure, one of two courses of action is taken depending on whether a condition is true or false

Aside: Fundamental Structures (4)

- In the repetition / iteration / loop structure, a group of statements is executed repeatedly until a condition becomes false

Conditional Statements

- Sometimes, one wants a statement to be executed only if some condition is satisfied
  - If this condition is not satisfied, either this statement should simply be skipped, or some other statement should be executed instead
- A conditional statement lets us choose which statement will be executed next
- Therefore, they are sometimes called selection statements
- Conditional statements give us the power to make basic decisions
- Java's main conditional statements are the if statement and the if-else statement
Part 1: Boolean Expressions

Boolean Expressions

- Like an arithmetic expression, a **boolean expression** is a combination of operators and operands, and it evaluates to a value.
- However, the type of the value a boolean expression evaluates to is not numeric, but `boolean (true or false)`.
- A boolean expression can be:
  - The comparison of two values using a comparison operator
  - A variable which has type `boolean`
  - A literal which has type `boolean (true or false)`
  - The negation of another boolean expression using the `!` operator
  - The combination of two or more other boolean expressions using the `&&` or `||` operators

Comparison Operators (1)

- Java's equality operators or comparison operators (also called *relational operators*) are used to compare numeric or character values:
  - `==`: equal to
  - `!=`: not equal to
  - `<`: less than
  - `>`: greater than
  - `<=`: less than or equal to
  - `>=`: greater than or equal to
- Both sides of each of these operators can be expressions

Comparison Operators (2)

- Equality (==) and inequality (!=) operators apply to values that have any type.
- The other comparison operators (<, >, <=, >=) only apply to values which have a numeric type (byte, short, int, long, float, double) or that have type `char`:
  - They do not apply to values that have type `boolean`.
- If the operands of a comparison operator have different types, the operand whose type has lower precision gets promoted to the other operand's type (via arithmetic promotion).
Comparison Operators (3)

• Even though the operands of a comparison operator may have various types, the type of the result of the comparison is always the same: boolean
  – This implies that the result of a comparison is always true or false

Comparison Examples (1)

• \((\text{denominator} == 0)\)
  – Evaluates to true if \(\text{denominator}\) is equal to 0, evaluates to false otherwise
• \((\text{denominator} != 0)\)
  – Evaluates to true if \(\text{denominator}\) is not equal to 0, evaluates to false otherwise
• \((\text{balance} < \text{amount})\)
  – Evaluates to true if the value of \(\text{balance}\) is strictly less than the value of \(\text{amount}\), evaluates to false otherwise
• \((\text{balance} > \text{amount})\)
  – Evaluates to true if the value of \(\text{balance}\) is strictly greater than the value of \(\text{amount}\), evaluates to false otherwise

Comparison Examples (2)

• \((\text{balance} <= \text{amount})\)
  – Evaluates to true if the value of \(\text{balance}\) is less than or equal to the value of \(\text{amount}\), evaluates to false otherwise
  – Note that using \(=\) will not work; the compiler will generate an error
• \((\text{balance} >= \text{amount})\)
  – Evaluates to true if the value of \(\text{balance}\) is greater than or equal to the value of \(\text{amount}\), evaluates to false otherwise
  – Again, note that using \(>=\) will not work; the compiler will generate an error

Comparison Operator Precedence

• Comparison operators have lower precedence than arithmetic operators, but higher precedence than the assignment operator

Therefore, the order of evaluation for this code fragment is the following (assume that \(a, c, d,\) and \(e\) have a numeric type):

\[
\text{boolean } b = a > c \times d + e;
\]

1. The product of \(c\) and \(d\) is evaluated first
2. Then, the value of \(c \times d\) is added to \(e\)
3. Then, the value of \(c \times d + e\) is compared to the value of \(a\)
4. Finally, the result of the comparison is stored in variable \(b\)
Comparison Operator Traps (1)

- Note the difference between the equality operator (==) and the assignment operator (=)
- The == operator compares two values for equality
  - Both sides can be expressions, so something like `count + 1 == max - 3` is perfectly legal
- The = operator assigns the value of the right-hand side to the variable on the left-hand side
  - The right-hand side can be any expression, but the left-hand side **MUST** be a variable
  - Therefore, something like `count + 1 = max - 3;` is illegal

Comparison Operator Traps (2)

- Using = instead of == is one of the most common errors, but the compiler will notice it if the operands are not of type boolean
- If both operands are of type boolean, using = instead of == may not be detected by the compiler in all cases!
  - In these cases, it will most likely result in a bug in your program
- Tips to avoid this error when operands are of type boolean:
  - When comparing the value of a boolean variable to a literal boolean value, put the literal on the left; that is, if `found` is a variable of type boolean, write `(true == found)` instead of `(found == true)`
  - `(true = found)` will produce a compilation error, but `(found = true)` will not

Comparison Operator Traps (3)

- Do not compare the value of boolean variables for equality with boolean literals
  - The expression `(found == true)`, where `found` is a variable of type boolean, is logically equivalent to `(found);` prefer the latter
  - Likewise, the expression `(found == false)` is equivalent to either `(found != true)` or `(!found);` use either of these instead
- In general, you should not compare the value of a variable of type boolean to a literal of type boolean; the preferred approach is the following:
  - If you want to check if the value stored in a variable `b` of type boolean is true, then the boolean expression `b` is enough
  - Likewise, if you want to check if the value stored in a variable `b` of type boolean is false, then the boolean expression `!b` is enough

Character Comparisons (1)

- We can use usual comparison / relational operators on character data
- The results are based on the Unicode character set
  - The Unicode character set assigns a number to each character
  - The numbers assigned to characters by the Unicode character set are used to perform the comparison
- The following expression evaluates to true because the number assigned to the character '+' by the Unicode character set is lower than the number assigned to the character 'J' by the same character set:

  ```java
  boolean lessThan = '+' < 'J';
  // lessThan has value true
  ```
Character Comparisons (2)

- In the Unicode character set, the numbers assigned to uppercase alphabetic characters ('A' - 'Z'), lower-case alphabetic characters ('a' - 'z') and digits ('0' - '9') not only follow the expected order, but are consecutive:
  - If 'A' is assigned the number $x$, then 'B' is assigned the number $x + 1$, 'C' is assigned the number $x + 2$, ...
  - If 'a' is assigned the number $y$, then 'b' is assigned the number $y + 1$, 'c' is assigned the number $y + 2$, ...
  - If '0' is assigned the number $z$, then '1' is assigned the number $z + 1$, '2' is assigned the number $z + 2$, ...

- Do not hesitate to use this property of characters in your programs.

Floating Point Comparisons (1)

- We also have to be careful when comparing two floating point values (float or double) for equality.
- You should rarely use the equality operator (==) when comparing two floating point values.
- In many situations, you might consider two floating point numbers to be "close enough" even if they aren't exactly equal.
- Therefore, to determine the equality of two floating point numbers, you may want to check if their difference is less than a certain threshold.

Floating Point Comparisons (2)

- The following code fragment is an example of comparing two floating point numbers by comparing their difference to a threshold:
  ```java
  // Assuming f1 >= f2
  final double THRESHOLD = 0.00001;
  difference = f1 - f2;
  boolean essentiallyEqual =
      difference < THRESHOLD;
  ```

String Equality (1)

- As mentioned before, the String type is a reference type, not a primitive type.
- Because of this:
  - The == and != operators do not work quite as you would expect with Strings; we will see what they do in detail later in the course.
  - The relational operators (<, >, <=, and >=) do not work at all with Strings; the compiler will produce an error if you try to use them with operands of type String.
- *Never use the == and != operators to compare Strings for equality (or inequality) unless you are really sure of what you are doing.*
  - This is a common source of bugs.
String Equality (2)

• Instead, to compare two Strings for equality, we use the `equals()` method.
• The boolean expression `s1.equals(s2)`, where `s1` and `s2` are both variables of type `String`, will compare the Strings stored in variables `s1` and `s2` to determine if they are equal.
• In a method call like `s1.equals(s2)`
  – the variable before `.compareTo` (in this case, `s1`) is called the target.
  – the expression between the parentheses (in this case, `s2`) is called the parameter.
• The expression `s1.equals(s2)` method will evaluate to `true` if and only if the target `String` is equal to the parameter `String`; it will evaluate to `false` otherwise.

String Equality (3)

– In the above expression, the target `String` is the one stored in `String` variable `s1`, and the parameter `String` is the one stored in `String` variable `s2`.
– Two `Strings` are equal if and only if they contain the same characters in the same order.
• To compare the `String` stored in a `String` variable with a `String` literal for equality, simply make the `String` literal the parameter `String` of the call to the `equals()` method.
  – For example, the expression `s1.equals("Hello")` will evaluate to `true` if and only if the `String` stored in variable `s1` (of type `String`) is equal to "Hello"; otherwise, it will evaluate to `false`.
• To compare two `Strings` for inequality, place a `!` in front of the call to the `equals()` method.

String Equality (4)

– For example, `!s1.equals(s2)` (where `s1` and `s2` are both variables of type `String`) will evaluate to `true` if the `String` stored in variable `s1` is not equal to the `String` stored in variable `s2`.
• Using the `equals()` method on `Strings` performs a case-sensitive comparison.
  – To perform a case-insensitive comparison instead, replace `equals()` by `equalsIgnoreCase()`.

String Comparisons (1)

• As mentioned previously, the relational operators do not work on `Strings` (or any other primitive types).
• To compare two `Strings` for order, we use the `compareTo()` method.
  • The expression `s1.compareTo(s2)`, where `s1` and `s2` are both variables of type `String`, will compare the `Strings` stored in variables `s1` and `s2` to determine which one occurs before the other in a predetermined lexicographical order.
    – This lexicographical order is based on the character codes of the individual characters of which each `String` consists.
String Comparisons (2)

- The expression `s1.compareTo(s2)` returns a value of type `int`, not `boolean`; this value is:
  - strictly less than 0 if the target String occurs before the parameter String (that is, if `s1` occurs before `s2`)
  - equal to 0 if the target String and the parameter String are equal (that is, `s1` and `s2` are equal)
  - strictly greater than 0 if the target String occurs after the parameter String (that is, if `s1` occurs after `s2`)

- To compare the String stored in a String variable with a String literal for order, simply make the String literal the parameter String of the call to the `compareTo()` method.

String Comparisons (3)

- For example, the value the expression `s1.compareTo("Hello")` evaluates to depends on whether the String stored in variable `s1` (of type `String`) occurs before, is equal to, or occurs after the String "Hello" in the lexicographical order used to compare Strings.

- Using the `compareTo()` method on Strings performs a case-sensitive comparison.
  - To perform a case-insensitive comparison instead, replace `compareTo()` by `compareToIgnoreCase()`.

- Note that the `equals()` and `compareTo()` methods do not work on primitive types.
  - The compiler will produce an error if you try to use them with primitive types.

Logical Operators

- Boolean expressions can also use the following logical operators:
  - Logical NOT: `!` (unary)
  - Logical OR: `||`
  - Logical AND: `&&`

- All three operators take operands of type `boolean` and produce results of type `boolean`.
- Logical NOT is a unary operator (it has one operand), but logical AND and logical OR are binary operators (they each have two operands).

Logical Operator Examples

```java
boolean choice = false;
boolean reverseChoice = !choice;  // Unary

boolean choice = !(x > 5);         // Unary with expression

boolean choice = (x > 5) && (y < 10);  // Binary with expressions
```
Logical NOT

- The **logical NOT** operation is also called *logical negation* or *logical complement*.
- The logical NOT operator is `!` in Java.
  - It is placed just before the boolean expression it negates, just like the unary `-` operator in arithmetic expressions.
- The `!` operator negates the value of a variable of type `boolean` or of a boolean expression.
  - If some boolean expression `a` evaluates to `true`, then `!a` evaluates to `false`.
  - If `a` evaluates to `false`, then `!a` evaluates to `true`.
  - Note that if `a` is a variable of type `boolean`, the `!` operator does not change the value of `a` (just like `-x` does not change the value of `x`).

Truth Tables

- The possible values of boolean expressions can be shown using **truth tables**.
- A truth table contains all possible combinations of values for the terms in the expression.
- The value of the expression for each combination is also shown.
- Below is the truth table for boolean expression `!a`:

<table>
<thead>
<tr>
<th><code>a</code></th>
<th><code>!a</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td><code>false</code></td>
<td><code>true</code></td>
</tr>
</tbody>
</table>

Logical AND and Logical OR

- A **logical AND** expression evaluates to `true` only if both `a` and `b` evaluate to `true`; it evaluates to `false` if `a`, or `b`, or both evaluate to `false`.
  - In Java, the logical AND operator is `&&`.
  - An example of a boolean expression in Java involving the `&&` operator is `a && b`.
- A **logical OR** expression evaluates to `true` if `a`, or `b`, or both evaluate to `true`; it evaluates to `false` only if both `a` and `b` evaluate to `false`.
  - In Java, the logical OR operator is `||`.
  - An example of a boolean expression in Java involving the `||` operator is `a || b`.

Truth Tables: Logical AND / OR

- As mentioned before, a truth table shows the possible `true` / `false` combinations of the terms.
- Because `&&` and `||` each have two operands, there are four possible combinations of `true` and `false`.

| `a`  | `b`  | `a && b` | `a || b` |
|------|------|----------|----------|
| `true` | `true` | `true`   | `true`   |
| `true` | `false` | `false`  | `true`   |
| `false` | `true` | `false`  | `true`   |
| `false` | `false` | `false`  | `false`  |
Logical Operator Precedence (1)

- Like arithmetic operators, logical operators have precedence rules among themselves.
- Logical operator `!` has higher precedence than logical operator `&&`, and logical operator `&&` has higher precedence than logical operator `||`.
- Consider the following expression (assume that a, b, and c all have type `boolean`):

  ```java
  a || b && !c
  ```

  1. First, the negation of c is evaluated
  2. Then, b is "AND-ed" with the value of !c
  3. Finally, a is "OR-ed" with the value of b && !c

Logical Operator Precedence (2)

- Logical operators also have precedence rules relative to other kinds of operators.
- Logical operators have lower precedence than comparison operators, but higher precedence than the assignment operator.
- Therefore, the order of evaluation for this code fragment is the following (assume that a has type `boolean`):

  ```java
  boolean b = a && c < d;
  ```

  1. First, the c and d are compared
  2. Then, the result of the comparison is "AND-ed" with the value of a
  3. Finally, the result of the logical operator is assigned to variable b

Short-Circuit Evaluation

- Logical operators `&&` and `||` are evaluated in short-circuit.
  - If one of the operand of a `&&` operator evaluates to `false`, the remaining operands are not evaluated.
    - There is no need to do so considering that a logical AND expression always evaluates to `false` as soon as one of its operands evaluates to `false`.
  - Likewise, if one of the first of a `||` operator evaluates to `true`, the remaining operands are not evaluated.
    - Again, there is no need to do so considering that a logical OR expression always evaluates to `true` as soon as one of its operands evaluates to `true`.
- In Java, logical operators are evaluated from left to right.

More on Truth Tables

- Specific expressions can be evaluated using truth tables as well:

<table>
<thead>
<tr>
<th>total &lt; MAX</th>
<th>found</th>
<th>!found</th>
<th>total &lt; MAX &amp;&amp; !found</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
Boolean Expressions: Exercises

- Write boolean expressions that evaluate to true if and only if the given conditions are true
  - The absolute value of variable $a$ (of type int) is greater than 100
    - $(a > 100) || (a < -100)$
  - The values of variables $a$, $b$, and $c$ are all different
    - $(a != b) && (b != c) && (a != c)$
  - The character stored in either variable $c1$, or variable $c2$ (both of type char), or both, is a digit
    - $((0' <= c1) && (c1 <= '9')) || ((0' <= c2) && (c2 <= '9'))$
  - The value stored in exactly one of the two variables $a$ and $b$ (both of type int) is equal to 0
    - $((a == 0) && (b != 0)) || ((b == 0) && (a != 0))$

Part 2: Conditional Statements

The if Statement

- The if statement has the following syntax:
  ```java
  if ( condition )
  statement;
  ```
  - The condition **MUST** be:
    - a boolean expression; therefore, it must evaluate to either true or false
    - surrounded by parentheses

  If the condition evaluates to true, then `statement` is executed; if it evaluates to false, then `statement` is skipped.

  - if is a Java reserved word

if Statement Example

- Here is an example of an if statement:
  ```java
  if (money > COST)
      money = money - COST;
      System.out.println("You have $" + money);
  ```
  - First, the condition `money > COST` is evaluated; the value of `money` is either greater than the value of `COST`, or it is not
  - If the condition `money > COST` evaluates to true, the assignment statement is executed; if it evaluates to false, the assignment statement is skipped
  - Either way, the call to `println()` is executed next
Logic of an \texttt{if} Statement

- If the \texttt{condition} is \texttt{false}, then \texttt{false} is returned.
- If the \texttt{condition} is \texttt{true}, then \texttt{true} is returned.
- The \texttt{statement} is executed.
- The \texttt{(rest of the program)} is executed.

The \texttt{if} Statement: Exercise

- Complete the \texttt{main()} method of the \texttt{BusRide} class by adding code to check whether the number of passengers is greater than the capacity of the bus.
  - If it is, then you should display a message asking for $x$ (where $x$ is the number of passengers in excess of the capacity of the bus) volunteers to travel in "economy class": on the roof.
  - Regardless of whether the number of passengers exceeds the capacity of the bus, you should display "Let's go!" after you have displayed whether or not volunteers are needed for "economy class".

BusRide.java

```java
import java.util.Scanner;

public class BusRide {
    public static void main (String[] args) {
        Scanner keyboard = new Scanner(System.in);
        final int CAPACITY = 56;
        int passengers;

        System.out.print("Enter the number of people that want " +
                         "to get on the bus: ");
        passengers = keyboard.nextInt();

        // Add your code here
    }
}
```

The \texttt{if-else} Statement

- An \texttt{else} clause can be added to an \texttt{if} statement to make it an \texttt{if-else} statement.
- An \texttt{if-else} statement has the following syntax:

  ```java
  if ( condition )
      statement1;
  else
      statement2;
  ```

  - If \texttt{condition} evaluates to \texttt{true}, \texttt{statement1} is executed but not \texttt{statement2}.
  - If \texttt{condition} evaluates to \texttt{false}, \texttt{statement2} is executed but not \texttt{statement1}.
  - \texttt{else} is also a reserved word in Java.
  - One of the two statements will be executed, but not both.
Logic of an \textit{if-else} Statement

\begin{center}
\begin{tikzpicture}
    \node (condition) {condition \textit{evaluated}};
    \node [below left of=condition, text width=1.5cm] (false) {false};
    \node [below right of=condition, text width=1.5cm] (true) {true};
    \node [below of=true, text width=0.5cm] (statement1) {statement1};
    \node [below of=false, text width=0.5cm] (statement2) {statement2};
    \node [below of=statement1, text width=5cm] (rest) {(rest of the program)};

    \draw [->] (condition) -- (false);
    \draw [->] (condition) -- (true);
    \draw [->] (false) -- (statement2);
    \draw [->] (true) -- (statement1);
    \draw [->] (true) -- (rest);
\end{tikzpicture}
\end{center}

The \textit{if-else} Statement: Exercise

- Complete the \texttt{main()} method of the \texttt{Wages} class by adding code to compute the gross earnings of an employee
  - If the employee has worked more than 40 hours during his / her work week, he / she should be paid 1.5 times his / her hourly wage for all hours worked in excess of 40
  - Can you rewrite your code using only a regular \texttt{if} statement (that is, one that does not have an \texttt{else} clause)?

\begin{verbatim}
import java.util.Scanner;
public class Wages {
    public static void main (String[] args) {
        Scanner keyboard = new Scanner(System.in);
        final double RATE = 8.25;  // Regular pay rate
        final int STANDARD = 40;   // Standard hours in a work week
        double pay = 0.0;
        int hours;
        System.out.print("Enter the number of hours worked: ");
        hours = keyboard.nextInt();
        // Your code here
    }
}
\end{verbatim}

Block Statements (1)

- Several statements can be grouped together into a block statement
- A block is delimited by braces ( \{ \} )
  - All statements between these braces are part of the block
- A block statement can be used wherever a single statement is called for in the Java syntax
- For example, in an \texttt{if} statement, the \texttt{if} portion could be a block statements
- In an \texttt{if-else} statement, the \texttt{if} portion, or the \texttt{else} portion, or both, could be block statements
If you want to put more than one statement in the if clause of an if or if-else statement, or the else clause of an if-else statement, you must group these statements in a block statement.

If you don't, you will get an error:
- Most often, it will be a semantic error; statements will be executed when they should not be, causing incorrect results or abnormal termination.
- It may be syntactic; for example, if you have multiple statements in the if clause of an if-else statement, the compiler will detect an else clause without an if.

For example, the following code fragment will cause a compilation error:
```java
if (condition)
    statement1;
    statement2;
else
    statement3;
```

When the compiler reaches statement2, it will assume that the if is part of an if statement (not an if-else statement), and that statement2 should be executed regardless of whether condition evaluates to true or false.

Then, when the compiler reaches the else, it will not be able to match it with any if statement, and thus will display an error.

What will happen if the following code fragment is executed:
```java
if (a < b)
    System.out.println(a);
    System.out.println(b);
```

The second println() call will be executed regardless of whether the condition evaluates to true or false.

In Java, syntax is what determines which clause a statement belongs to:
- Indentation has nothing to do with this
- Tip #1: Always use block statements with if and if-else statements, even when the block statement contains only one statement
- Tip #2: Always have consistent indentation
Block Statements: Exercise

• Complete the main() method of the GuessGame class by adding code to determine whether the user won or not
  – The player wins if he/she is able to guess the number that the program chose at random
  – If the player wins, you should display a message stating that he/she has won, and the amount of money he/she has won
  – If the player loses, you should display a message stating that he/she has lost, what the number chosen by the program was, and the amount the player has lost
  – Whether the player wins or loses, you should display the amount of money he/she has after playing the game

import java.util.Scanner;
import java.util.Random;

public class GuessGame {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        Random randomSource;
        final int UPPER_BOUND = 10;
        double money;
        double betAmount;
        int myNumber;
        int yourNumber;
        randomSource = new Random();
        System.out.print("How much money do you have? ");
        money = keyboard.nextDouble();
        // Continued on next slide
    }
}

// Continued from previous slide
System.out.print("How much would you like to bet? ");
betAmount = keyboard.nextDouble();

myNumber = randomSource.nextInt(UPPER_BOUND) + 1;
System.out.print("I've chosen a number between 1 and " + UPPER_BOUND + ". Try to guess it: ");
yourNumber = keyboard.nextInt();

// Add your code here
}

Nested if Statements (1)

• The statement executed as a result of the if clause of an if or if-else statement, or the else clause of an if-else statement, could itself be another if or if-else statement
• These are called nested if statements
• Indentation does not determine which if keyword matches with which else keyword (if any) or vice-versa; it is determined by syntax (that is, the order of the clauses, or the presence of braces {})
  – An else clause matches with the nearest if
  – But, if you make all the if and else clauses block statements and indent well, you will never make a mistake

// Continued on next slide
System.out.print("How much money do you have? ");
money = keyboard.nextDouble();

// Continued on next slide
System.out.print("How much would you like to bet? ");
betAmount = keyboard.nextDouble();

myNumber = randomSource.nextInt(UPPER_BOUND) + 1;
System.out.print("I've chosen a number between 1 and " + UPPER_BOUND + ". Try to guess it: ");
yourNumber = keyboard.nextInt();

// Add your code here
}
Nested if Statements (2)

• One can write nested if-else statements like this:

```java
if (condition1)
  if (condition2)
    statement1;
  else
    statement2;
else
  if (condition3)
    statement3;
  else
    statement4;
```

Logic of Nested if Statements

![Diagram showing the logic of nested if statements]

Nested if Statements: Exercise

• Complete the `main()` method of the `MinOfThree` class by adding code which determines which of the three numbers entered by the user is the smallest number, and displays that number.

• Can you write this code both with and without using block statements?

```java
import java.util.Scanner;

public class MinOfThree {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int num1, num2, num3, min;
    System.out.print("Enter a number: ");
    num1 = keyboard.nextInt();
    System.out.print("Enter another number: ");
    num2 = keyboard.nextInt();
    System.out.print("Enter a third number: ");
    num3 = keyboard.nextInt();
    // Add your code here
  }
}
```
Multiple Execution Branches (1)

• Nested if/else statements are necessary when there are more than two branches/conditions/cases.

• One can handle multiple branches by using the following idiom:

```java
if (condition1) {
    // Code for condition 1
} else if (condition2) {
    // Code for condition 2
} else if (condition3) {
    // Code for condition 3
    // ...
} else {
    // Everything else, if needed
}
```

Multiple Execution Branches (2)

• This is equivalent to:

```java
if (condition1) {
    // Code for condition 1
} else if (condition2) {
    // Code for condition 2
} else if (condition3) {
    // Code for condition 3
    // ...
} else {
    // Everything else, if needed
}
```

• But ensure your conditions are correct!

Multiple Branches: Exercise (1)

• Complete the main() method of the Drinks class by adding code that does the following:
  - If the user indicates that he/she wants to drink orange juice, the program should display two messages: "Vitamin C!" and "Your bones will thank you."
  - If the user indicates that he/she wants to drink milk, the program should display "Your bones will thank you."
  - If the user indicates that he/she wants to drink water, the program should display "The classics never die."
  - If the user indicates that he/she wants to drink wine, the program display a prompt asking him/her whether he wants red or white wine, and read the answer
    - The answer the program reads from the keyboard should be of type String

Multiple Branches: Exercise (2)

• If the user chooses red wine (by typing "red"), the program should display "Good for your heart."
• If the user chooses white wine (by typing "white"), the program should display "Good for your lungs."
• If the user enters an invalid option, the program should display "What kind of strange-colored wine is this?"
  - Note that the case of the user's answer does not matter; that is, "RED" should be considered to be the same as "red"
• If the user indicates that he/she wants beer, the program should display "Watch that belly!"
• If the user enters an invalid option, the program should display "That's not going to quench your thirst..."
Drinks.java

```java
import java.util.Scanner;

public class Drinks {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    int choice;
    System.out.println("Here is the drinks menu : ");
    System.out.println("1. Orange juice");
    System.out.println("2. Milk");
    System.out.println("3. Water");
    System.out.println("4. Wine");
    System.out.println("5. Beer");
    System.out.print("What will it be ? ");
    choice = keyboard.nextInt();
    // Add your code here
  }
}
```

Part 3: Exercises

Exercises (1)

1. Write a program which consists of a single class called OldEnough that asks the user for their age and displays "You can enter" if he/she is over 18, but displays "Sorry, you are not allowed here" otherwise.

2. Write a program which consists of a single class called BuyStuff that asks the user for two amounts, adds them, calculates tax at 15%, shows the result to the user, and asks for money. It then compares if the person gave enough money. If so, it displays the amount of change to return otherwise it displays a message asking for more money.

Exercises (2)

3. Write a program which consists of a single class called Calculator. This program will display a menu with the following options: (1) add, (2) subtract, (3) multiply, (4) divide (5) mod, and (6) quit. If the user enters something else, the program should display an error message. Otherwise, it should ask the user for two numbers, perform the calculation, and display the result.

4. Write a program which consists of a single class called SortThree. This program should ask the user to enter 3 integers. The program will then display these in increasing order.