STUDENT OUTLINE

Lesson 8 – Structured Programming, Control Structures, if-else Statements, Pseudocode

INTRODUCTION:

This lesson is the first of four covering the standard control structures of a high-level language. Using such control structures requires the creation of simple to complex Boolean statements that evaluate to true or false. After covering the relational and logical operators available in Java, the remainder of this lesson will present the **if-else** control structure.

The key topics for this lesson are:

A. Structured Programming
B. Control Structures
C. Algorithm Development and Pseudocode
D. Relational Operators
E. Logical Operators
F. Precedence and Associativity of Operators
G. The **if-else** Statements
H. Compound Statements
I. Nested **if-else** Statements
J. Conditional Operator
K. Boolean Identifiers

VOCABULARY:

STRUCTURED PROGRAMMING	CONTROL STRUCTURE
ALGORITHM	ITERATION
LOGICAL OPERATOR	PSEUDOCODE
IF-ELSE	RELATIONAL OPERATOR
STEPWISE REFINEMENT	COMPOUND STATEMENT
CONDITIONAL OPERATOR	BOOLEAN IDENTIFIER

DISCUSSION:

A. <u>Structured Programming</u>

1. Up to this point in your study of computer science and Java, you have created programs that used only sequential execution. So far most programs have consisted of a sequence of lines that are executed once, line-by-line. As we add the power of loops and selection, we need to use these tools in a disciplined manner.

DON'T USE THIS... EVER

2. In the early days of programming (1960's), the approach to writing software was relatively primitive and ineffective. Much of the code was written with `goto` statements that transferred program control to another part of the code. Tracing this type of code was an exercise in jumping from one spot to another, leaving behind a trail of lines similar to spaghetti. The term "spaghetti code" comes from trying to trace code linked together with `goto` statements.

3. The research of Bohm and Jacopini[1] has led to the rules of structured programming. Here are five tenets of structured programming. There are only three necessary control structures needed to write programs: sequence, selection, and iteration.

   a. No `goto` statements are to be used in writing code.
   b. All programs can be written in terms of three control structures: sequence, selection, and iteration.
   c. Each control structure has one entrance point and one exit point. We will sometimes allow for multiple exit points from a control structure using the `break` statement.
   d. Control structures may be stacked (sequenced) one after the other.
   e. Control structures may be nested inside other control structures. ONE WITHIN ANOTHER

4. The control structures of Java encourage structured programming. Staying within the guidelines of structured programming has led to great productivity gains in the field of software engineering.

B. Control Structures

See Handout H.A.8.2, *Control Structures in Java.*

1. There are only three necessary control structures needed to write programs: sequence, selection, and iteration.

2. Sequence refers to the line-by-line execution as used in your programs so far. The program enters the sequence, does each step, and exits the sequence.

OPTIONS ARE SELECTED AT RUN-TIME BASED ON CONDITIONS PRESENT

3. Selection is the control structure that allows choice among different directions. Java provides different levels of selection:

   - One-way selection with an `if` structure
   - Two-way selection with an `if-else` structure
   - Multiple selection with a `switch` structure *WONT COVER *

LOOPS WILL REPEAT SEGMENTS OF CODE UNTIL SOME CONDITION IS MET

4. Iteration refers to looping. JAVA provides three loop structures:

   - `while` loops
   - `do-while` loops
   - `for` loops
   - for-each Loop (later)

5. Of the seven control structures, the `if-else` and `while` loop are the most flexible and powerful for problem-solving. The other control structures have their place, but `if-else` and `while` are the most common control structures used in Java code.
6. The diagrams in H.A.8.2, *Control Structures in Java*, are flowcharts that describe the flow of program control. A statement rectangle in any control structure can be a simple line or even another control structure. A statement can also be a compound statement that consists of multiple statements.

[1] Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules, "*Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336-371."

FAKE CODE
·PLANNING
TOOL

C. Algorithm Development and Pseudocode

1. An algorithm is a solution to a problem. Computer scientists are in the problem-solving business. They use techniques of structured programming to develop solutions to problems. Algorithms will range from the easier "finding the average of two numbers" to the more difficult "visiting all the subdirectories on a hard disk, searching for a file."

2. A major task of the implementation stage is the conversion of rough designs into refined algorithms that can then be coded in the implementation language of choice.

NOT CODE!!

3. Pseudocode refers to a rough-draft outline of an answer, written in English-like terms. We will probably use phrases and words that are close to programming languages, but avoid using any specific language. Once the pseudocode has been developed, translation into code occurs more easily than if we had skipped this pseudocode stage.

4. Stepwise refinement is the process of gradually developing a more detailed description of an algorithm. Problem solving in computer science involves overall development of the sections of a program, expanding each section with more detail, later working out the individual steps of an algorithm using pseudocode, then finally writing a code solution.

DECOMPOSE THE PROBLEM

*SEE NEXT PAGE

See Handout H.A.8.3, *Pseudocode and Algorithm Development.*

5. The handout, H.A.8.3, *Pseudocode and Algorithm Development*, will present a thorough example of this process. You should read this now.

"gets"
= ASSIGNMENT

== RELATION
"equals"

D. Relational Operators

1. A relational operator is a binary operator that compares two values. The following symbols are used in Java as relational operators:

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

≤ D.N.E. IN CODE

2. A relational operator is used to compare two values, resulting in a relational expression. For example:

```
number > 16          grade == 'F'          passing >= 60
```

Development of pseudocode:

Stepwise refinement 1 - Overall sections of this problem:

I Get data from user
II Solve math
III Print answer

VERY GENERAL!!

Stepwise refinement 2 - More detailed pseudocode version:

I
A) Prompt user for three dimensions
B) Prompt user for weight

II
A) Determine longest of three dimensions
B) Calculate the girth using the other two dimensions

If package is too big and too heavy, print appropriate message
else if package is too big, print appropriate message
else if package is too heavy, print appropriate message
else print package is acceptable

EACH REFINEMENT ADD SPECIFICITY

Stepwise refinement 3 - Determining longest of three dimensions:

WE'LL TALK MORE ABOUT THIS SOON!!

3. The result of a relational expression is a **boolean** value, **true** or **false**.

4. When character data is compared, the ASCII code values are used to determine the answer. The following expressions result in the answers given:

`'A' < 'B'`	evaluates as **true**, (65 < 66)
`'d' < 'a'`	evaluates as **false**, (100 < 97)
`'t' < 'X'`	evaluates as **false**, (116 < 88)

In the last example, you must remember that upper case letters come first in the ASCII collating sequence; the lower case letters follow after and consequently have larger ASCII values than do upper case ('A' = 65, 'a' = 97).

E. Logical Operators

1. The three logical operators of programming are AND, OR, and NOT. These operators are represented by the following symbols in Java:

AND	`&&`
OR	`||` (two vertical bars)
NOT	`!`

A.K.A. DOUBLE PIPES

TRUTH TABLES

&&	T	F
T	T	F
F	F	F

2. The `&&` (and) operator requires both operands (values) to be true for the result to be true.

```
T and T = true
T and F = false
F and T = false
F and F = false
```

3. The following are Java examples of using the `&&` (and) operator.

`( (2 < 3) && (3.5 > 3.0) )`	evaluates as true
`( (1 == 0) && (2 != 3) )`	evaluates as false

*IMPORTANT IDEA!

The `&&` operator performs short-circuit evaluation in Java. If the first half of an `&&` statement is false, the operator immediately returns false without evaluating the second half.

||	T	F
T	T	T
F	T	F

4. The `||` (or) operator requires only one operand (value) to be true for the result to be true.

```
T or T = true
T or F = true
F or T = true
F or F = false
```

THERE IS A SPECIAL TYPE KNOWN AS XOR (EXCLUSIVE OR) WHERE T || T ⇒ F

WE WONT BE WORKING WITH THIS IN OUR YEAR

OFTEN REFERRED TO AS A CONDITIONAL STATEMENT

5. The following is a Java example of using the || (or) operator.

   ```
   ( (2+3 < 10) || (21 > 19) )
   ```
   evaluates as true

   The || operator also performs short-circuit evaluation in Java. If the first half of an || statement is true, the operator immediately returns true without evaluating the second half.

6. The ! operator is a unary operator that changes a **boolean** value to its opposite.

   ! false = true
   ! true = false

   NEGATION

F. Precedence and Associativity of Operators

1. Introducing two new sets of operators (relational and logical) adds to the complexity of operator precedence in Java. An abbreviated precedence chart is included here.

COMPARISON

LOGIC

Operator	Associativity
! unary - ++ --	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
&& (and)	left to right
|| (or)	left to right
= += -= *= /=	right to left

Table 8-1 Precedence and Associativity of Operators

2. Because the logical operators have low precedence in Java, parentheses are not needed to maintain the correct order of solving problems. However, they can be used to make complex expressions more readable.

ADD ( ) FOR CLARITY TO HUMAN READER

```
((2 + 3 < 10) && (75 % 12 != 12))     // easier to read
(2 + 3 < 10 && 75 % 12 != 12)         // harder to read
```

G. The **if-else** Statements

1. The general syntax of the **if-else** statement is as follows:

WHEN EXPRESSION IS TRUE

WHEN THE CONDITIONAL IS FALSE

```
if (expression)
   statement1;
else
   statement2;
```

A.K.A. "THE CONDITIONAL"

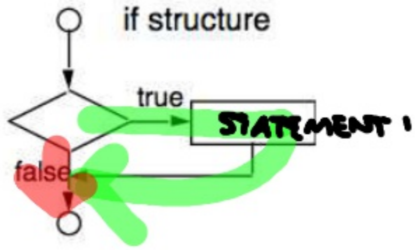
F?

2. **`if`** statements may omit the **`else`** option if it results in one-way selection.

SKIP STATEMENT1

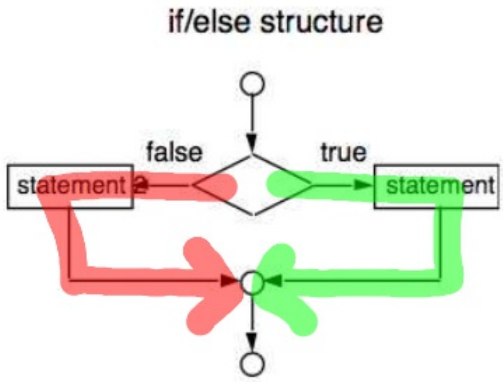
```
if (expression)
    statement1;
```

If the expression is non-zero, statement is executed, otherwise nothing is executed. The following flowchart illustrates the flow of control.



3. The full **`if-else`** statement allows for two-way control. If the value of the expression is true, statement1 is executed. If the value of the expression equals false, the **`else`** option results in statement2 being executed. The following flowchart from handout, H.A.8.2, illustrates the flow of control.

CODE ON PREVIOUS PAGE



4. The expression being tested must always be placed in parentheses. This is a common source of syntax errors.

CONDITIONAL

H. Compound Statements

1. The statement executed in a control structure can be a block of statements, grouped together into a single compound statement.

2. A compound statement is created by enclosing any number of single statements by braces as shown in the following example:

PREFERRED FORMATTING

```
if (expression)
{
  statement1;
  statement2;
  statement3;
}
else
{
  statement4;
  statement5;
  statement6;
}
```

T

F

I. Nested **if-else** Statements

1. The statement inside of an **if** or **else** option can be another **if-else** statement. Placing an **if-else** inside another is known as nested **if-else** constructions. For example:

PAIRED STATEMENTS SHOULD BE AT THE SAME INDENTATION LEVEL

```
if (expression1)
  if (expression2)
    statement1;   T2
  else
    statement2;   F2
else
  statement3;
```

T1 F1

2. The **else** option will be paired with the nearest unpaired **if**. `statement2` is the alternative action of the inner **if**, while `statement3` is the alternative action of the outer **if**.

3. The above example has three possible different outcomes as shown in the following chart:

expression 1	expression2	statement executed
true	**true**	statement1
true	**false**	statement2
false	**true**	statement3
false	**false**	statement3

```
if(exp1)
{ if(exp2)
    S1;
}
else
  S2;
```

FIX?

4. Caution must be shown when using **else** statements inside of nested **if-else** structures. For example:

```
if (expression1)
  if (expression2)
    statement1;
else
  statement2;
```

```
if (exp1)
{ if (exp2)
    S1
  else
    S2
}
```

Indentation is ignored by the compiler, hence it will pair the **else** statement with the inner **if**. If you want the **else** to get paired with the outer **if** as the indentation indicates, you need to add braces:

```
if (expression1)
{
  if (expression2)
    statement1;
}
else
  statement2;
```

DELIMITERS

The braces allow the else statement to be paired with the outer **if**.

EQUIVALENT? TRUE? ????

5. Another alternative to the example in Section 4. makes use of the && operator. A pair of nested **if** statements can be coded as a single compound && statement.

```
if (expression1 && expression2)
  statement1
else
  statement2;
```

6. The most common and effective use of nested **if-else** statements is called an **if-else** chain. See the following formatting styles:

Formatting style 1

```
if (expression1)
  statement1;
else
  if (expression2)
    statement2;
  else
    if (expression3)
      statement3;
    else
      statement4;
```

EXIT STRUCTURE

Formatting style 2

```
if (expression1)
  statement1;
else if (expression2)
  statement2;
else if (expression3)
  statement3;
else
  statement4;
```

I HATE THIS

Notice that each successive **if-else** statement is buried deeper in the overall structure. `statement4` will only be executed if the first three expressions are **false**.

7. Formatting style 1 lines up the **if** with its counterpart **else** keyword. This makes it easy to check syntax but the indentation can get rather deep. Formatting style 2 is a more compact version but you need to be careful about which **else** statement belongs to which **if**. Formatting style 1 is more appropriate if the statements are compound statements. Formatting style 2 is appropriate if the statements are single-line statements.

8. The advantage of such an **if-else** chain is efficiency in execution. If a **true** value is encountered at any level, that statement is executed and the rest of the structure is ignored.

9. Consider the following example of determining the type of triangle given the three sides A, B, and C.

```
if ( (A == B) && (B == C) )
  System.out.println("Equilateral triangle");
else  if ( (A == B) || (B == C) || (A == C) )
  System.out.println("Isosceles triangle");
else
  System.out.println("Scalene triangle");
```

if((A!=B)&&(B!=C)
&&(A!=C))

EFFICIENT TO
REAL WORLD
PROBABILITIES

If an equilateral triangle is encountered, the rest of the code is ignored. Such a chain is best constructed by placing the most demanding case at the top and the least demanding case at the bottom.

J. Conditional Operator (optional) TERNARY OPERATOR

1. Java provides an alternate method of coding an **if-else** statement using the conditional operator. This operator is the only ternary operator in Java, as it requires three operands. The general syntax is:

T

```
(condition) ? statement1 : statement2;
```

F

(x>7)?x++:x--;

2. If the condition is true, statement1 is executed. If the condition is false, statement2 is executed.

ONLY FOR
SIMPLE THINGS!!

3. This is appropriate in situations where the conditions and statements are fairly compact.

```
int max(int a, int b)  // returns the larger of two
//integers
{
  (a > b) ? return a  :  return b;
}
```

A.K.A.
if(x>7)
  x++;
else
  x--;

K. Boolean Identifiers

1. The execution of **if-else** statements depends on the value of the Boolean expression. We can use **boolean** variables to write code that is easier to read.

2. For example, the **boolean** variable done could be used to write code that is more English-like.

   Instead of

```
if (done == true)
  System.out.println("We are done!");
```

IDENTICAL

   we can write

```
if (done)
  System.out.println("We are done!");
```

3. Using Boolean identifiers with conditional loops allows a separation of solving expressions from thinking about program control. Here is an example solution using the **while** control structure (to be covered in the next lesson), presented in a blend of Java and pseudocode:

WE'LL DISCUSS THIS MORE!!..

```
boolean done = false;

while (!done)
 // do some code that could change the value of done
```

4. Where appropriate you are encouraged to use **boolean** variables to aid in program flow and readability.

SUMMARY/ REVIEW: Control structures are a fundamental part of high level languages. Fluency in any high level language only comes with practice. You will need to practice control structures in Java as well as all other aspects of the language.

ASSIGNMENT: Lab Exercise, L.A.8.1, *IRS*