STUDENT LESSON

A8 – Control Structures if, if-else, switch

STUDENT LESSON

A8 – Control Structures
if, if-else, switch

INTRODUCTION: Any sort of complex program must have some ability to control flow. Without this control, programs become limited to one basic job each time the program is run. The most basic of these control structures is the **if** statement, followed by the **if-else**, and then the **switch** statement.

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
L. Switch Statements (Optional)

VOCABULARY:

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

DISCUSSION:

A. Structured Programming

1. 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 line in 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. The complexity this added to code led to the development of structured programming.

2. The research of Bohm and Jacopini[1] has led to the rules of structured programming. Here are five tenets of structured programming.

   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.

3. 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

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. This allows for sequences to do only a limited job during each execution.

3. Selection is the control structure that allows choice among different paths. 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

4. Iteration refers to looping. Java provides three loop structures. These will be discussed in length in Student Lesson A12.

   - **`while`** loops
   - **`do-while`** loops
   - **`for`** loops

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."

---

[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."

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.

3. Pseudocode refers to a rough-draft outline of an answer, written in English-like terms. These generally use phrases and words that are close to programming languages, but avoid using any specific language syntax. 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, and then finally writing a code solution.

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

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

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

3. The result of a relational expression is a **`boolean`** value of either **`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 in the AP subset are AND, OR, and NOT. These operators are represented by the following symbols in Java:

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

   These logical operators allow us to combine conditions. For example, if a dog is gray and weighs less than 15 pounds it is the perfect lap dog.

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

```
(true  && true) -> true
(true  && false) -> false
(false && true)  -> false
(false && false) -> false
```

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

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

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

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

```
(true  || true)  -> true
(true  || false) -> true
(false || true)  -> true
(false || false)  -> false
```

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

   `((2+3 < 10) || (19 > 21))` -> 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
   (! true == false) -> true
   (! true == true) -> false

!(2 < 3) -> false

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.

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.

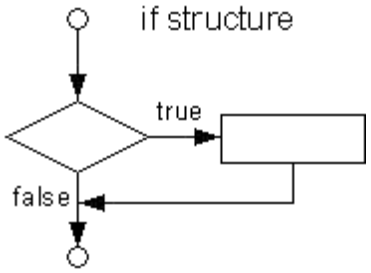
```
((2 + 3 < 10) && (75 % 12 != 12))    // easier to read
(2 + 3 < 10 && 75 % 12 != 12)        // harder to read
```

G. The **if-else** Statements

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

```
if (expression){
   statement1;
}
```

If the expression evaluates to true, statement1 is executed. If expression is false then nothing is executed and the program execution picks up after the ending curly brace (}). The following diagram shows the flow of control:

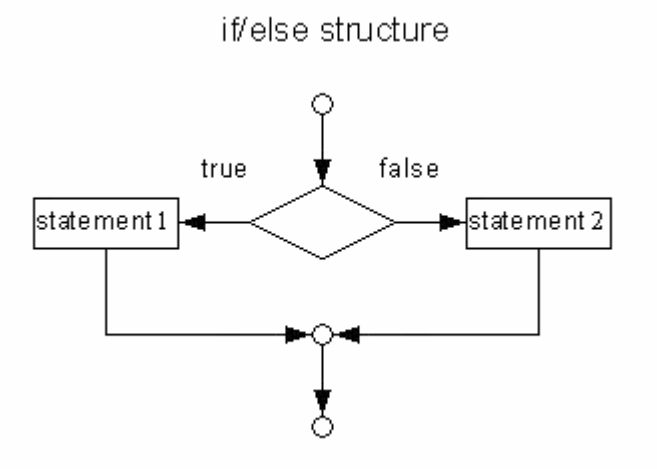


2. To provide for two-way selection an **if** statement may add an **else** option.

```
if (expression){
```

```
   statement1;
}else{
   statement2;
}
```

If the expression evaluates to true, the statement is executed. In an if-else statement, if the expression is false then statement2 would be executed. The following flowchart illustrates the flow of control.



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

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:

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

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:

```
if (expression1){
  if (expression2){
```

```
    statement1;
  }else{
    statement2;
  }
}else{
  statement3;
}
```

2. Here, your braces will need to be correct to ensure that the ifs and elses get paired with their partners.

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	**not tested**	statement3

4. Technically, braces are not needed for if and if-else structures if you only want one statement to execute. However, caution must be shown when using **else** statements inside of nested **if-else** structures. For example:

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

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;
```

Important Concept

The braces allow the else statement to be paired with the outer **`if`**. However, if you always use braces when writing **if** and **if-else** statements, you will never have this problem.

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. Both of these blocks of code would have the exact same effect, but the second one is slightly easier to read.

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

//or...
```

```
if (expression1 && expression2){
  statement1;
}
```

The second block of code makes the conditions clearer to another programmer.

6. 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 an equilateral triangle is encountered, the rest of the code is ignored. This can help to reduce the execution time of a program.

J. Conditional Operator (optional)

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:

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

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

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;
}
```

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 reads more like English.

Instead of

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

```
}
```

we can write

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

3. Programmers often use **`boolean`** variables to aid in program flow and readability. The second version is the more preferred way of using a **`boolean`** variable in this situation because it is less dangerous. If you make a mistake and only put = instead of == Java will not catch that and interprets the statement as assignment. Some strange results could occur and it can take the programmer a while to catch the error.

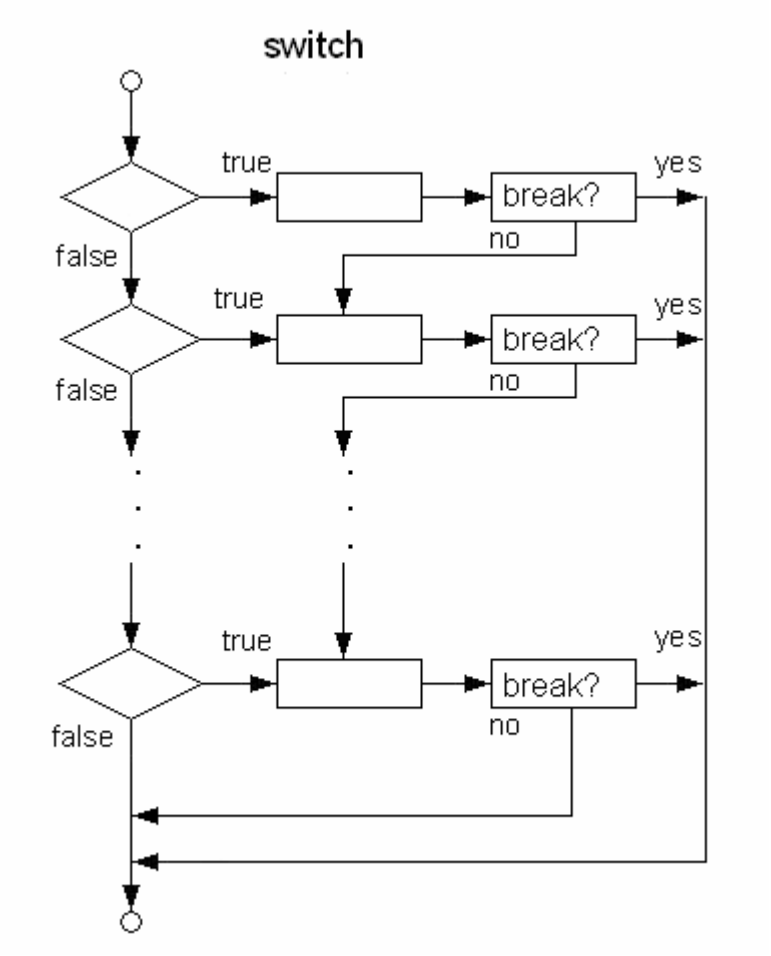
L. Switch Statements (optional)

1. Consider a simple user menu for a store simulation program. There should be options to buy certain items, check your total money spent, cancel items selected, exit, and finish and pay. We could take the input from this menu and do a complicated, nested series of **`if-else`** statements, but that would quickly become bulky and difficult to read. However, there is an easy way to handle such data input with a **`switch`** statement. Depending on which command is chosen, the program will select one direction out of many. The AP exam does not test on the **`switch`** statement, but we include it here at the end of this chapter because it is a very useful tool to have in your programming toolkit.

2. The general form of a **`switch`** statement is:

```
switch (expression){
  case value1:
      statement1;
      statement2;
      ...
      break;
  case value2:
      statement3;
      statement4;
      ...
      break;
  case valuen: //any number of case statement may be used
      statement;
      statement;
      break;
  default:
      statement;
      statement;
      break;
}        /* end of switch statement */
```

3. The flow of control of a **`switch`** statement is illustrated in this diagram:



4. The **`switch`** statement attempts to match the integer value of the expression with one of the **`case`** values.

5. If a match occurs, then all statements past the **`case`** value are executed until a **`break`** statement is encountered.

6. The effect of the **`break`** statement causes program control to jump to the end of the **`switch`** statement. No other cases are executed.

7. A very common error when coding a **`switch`** control structure is forgetting to include the **`break`** statements to terminate each case value. If the **`break`** statement is omitted, all the statements following the matching **`case`** value are executed. This is usually very undesirable.

8. If it is possible that none of the case statements will be true, you can add a **`default`** statement at the end of the **`switch`**. This will only execute if none of the case statements happened. If all possibilities are covered in your case statements, the **`default`** statement is unnecessary. Note that the **`default`** statement can actually be placed anywhere. If you place the **`default`** in the beginning or middle of the **`switch`**, you will probably want to end the

default case with a **break**. Otherwise, execution will continue with the case after the **default**.

```
int i = 4;
switch (i) {
   case 1: System.out.println("Apple"); break;
   default: System.out.println("Orange");
   case 2: System.out.println("Banana"); break;
}

Orange
Banana
```

9. The following example applies the **switch** statement to printing the work day of the week corresponding to a value. We pass in the integer day:

```
switch (day){
  case 1:  System.out.println ("Monday");  break;
  case 2:  System.out.println ("Tuesday");  break;
  case 3:  System.out.println ("Wednesday");  break;
  case 4:  System.out.println ("Thursday");  break;
  case 5:  System.out.println ("Friday");  break;
  default:  System.out.println ("not a valid day");  break;
}
```

10. Suppose we wanted to count the occurrences of vowels and consonants in a stream of text.

```
if (('a' <= letter) && (letter <= 'z')){
  switch (letter){
    case 'a' :  case 'e' :  case 'i' :  case 'o' :  case 'u' :
      vowel++;
      break;
    default :
      consonant++;
      break;
  }
}
```

    a. Note that multiple **case** values can lead to one set of statements.
    b. It is good programming practice to include a **break** statement at the end of the **switch** structure. If you need to go back and add another **case** statement at the end of the **switch** structure, a **break** statement already terminates the previous case statement and there is no chance that you might forget to add a break statement.

11. There are programming situations where the **switch** statement should not replace an **if-else** chain. If the value being compared must fit in a range of values, the **if-else** statement should be used.

```
if(score >= 90 && score <= 100){
  grade = 'A';
}else if{(score >= 80 && score < 90)
  grade = 'B';
}else if{(score >= 70 && score < 80)
  grade = 'C';
```

```
}
```

etc...

You should not replace the above structure with a **`switch`** statement.

12. Finally, the **`switch`** statement cannot compare **`double`** values.

SUMMARY/ REVIEW: Control structures are a fundamental part of Java. You will need to practice control structures in Java to become familiar with what types of situations they are useful in. Also, Boolean expressions are very useful and should be used whenever appropriate to make coding easier.

ASSIGNMENT: Lab Assignment A8.1, *CheckMail*
Lab Assignment A8.2, *IRS*