3.1 Special Operators

Increment and Decrement Operator

The auto-increment and auto-decrement operators are arithmetic and operate on one operand. The *auto-increment operator* (++) adds 1 to its operand. The *auto-decrement operator* (--) subtracts 1 from its operand. The statement

```
count++;
```

is functionally equivalent to

```
count = count + 1;
```

Assignment Operator

Often we perform an operation on a variable, and then store the result back into that variable**.** Java provides *assignment operators* to simplify that process**.** For example, the statement

```
num += count;
```

is equivalent to

```
num = num + count;
```

There are many assignment operators, including the following:

Operator	Example	Meaning
=	x = y	simple assignment
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

The right hand side of an assignment operator can be a complex expression. The entire right-hand expression is evaluated first, then the result is combined with the original variable. Therefore:

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```

The behavior of some assignment operators depends on the types of the operands. If the operands to the += operator are strings, the assignment operator performs string concatenation. The behavior of an assignment operator (+=) is always consistent with the behavior of the "regular" operator (+).

```
String s1 = "Hi";
s1 += 3;
System.out.println (s1);
s1 += (4 + 5);
System.out.println (s1);
```

Operator Precedence

highest precedence →	(1)	`!, ++, --`
	(2)	`*, /, %`
	(3)	`+, -`
	(4)	`<, >, <=, >=`
	(5)	`==, !=`
	(6)	`&&`
	(7)	`||`
lowest precedence →	(8)	`=, +=, -=, *=, /=, %=`

3.2 Boolean Expressions

Flow of control

Unless specified otherwise, the order of statement execution through a method is *linear*: one statement after the other in sequence. Some programming statements modify that order, allowing us to:

- decide whether or not to execute a particular statement, or
- perform a statement over and over, repetitively or iteratively

These decisions are based on a *boolean expression* (also called a *condition* that evaluates to *true* or *false* The order of statement execution is called the *flow of control.*

A *condition* often uses one of Java's *equality operators* or *relational operators*, which all return *boolean* (`true` or `false`) results:

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

Note the difference between the *equality operator* (==) and the *assignment* operator (=)

```
int x = 3;

x == 4;

x = 4;
```

Example 1: What will be output by the following statement?

```
System.out.println (5 + 3 < 6 - 1);
```

Output:

Logical Operators

Boolean expressions can use the following *logical operators*:

`!` Logical NOT `&&` Logical AND `||` Logical OR

They all take boolean *operands* and produce boolean *results*. The *logical AND* expression `a && b` is true if both `a` and `b` are true, and false otherwise. The *logical OR* expression, `a || b,` is true if `a` or `b` or both are true, and false otherwise.

Truth Tables – Fill out the following "truth table" if `a` and `b` are both boolean values.

`a && b`

&&	T	F
T		
F		

`a || b`

||	T	F
T		
F		

`!a`

!	
T	
F	

Example 2: Rewrite each condition below as a valid Java statement (use a `boolean` expression) and assign the result to the already declared `boolean` variable `result`:

a. x > y > z

b. x and y are both less than 0

c. neither x nor y is less than 0

d. x is equal to y but not equal to z

De Morgan's Law:

```
NOT (P or Q) = (NOT P) and (NOT Q)          !(P || Q) == !P && !Q

NOT (P and Q) = (NOT P) or (NOT Q)          !(P&&Q) == !P || !Q
```

Example 3: Assume that `a` and `b` have been defined and initialized as `int` values. Simplify the following statement using De Morgan's Law:

```
! ( !(a != b) && (b > 7) )
```

Short circuited operators

The processing of logical AND and logical OR is "*short circuited*" meaning if the left operand is sufficient to determine the result, the right operand is NOT evaluated.

```
boolean result = count != 0 && total/count > MAX;
```

3.3 The `if` Statement

Conditional statements

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. Some conditional statements in Java are:

- the `if` *statement*
- the `if-else` *statement*

`if` statement

The `if` *statement* has the following syntax:

```
if ( condition )
   statement;
```

`if-else` statement

An `else` *clause* can be added to an `if` statement to make an `if-else` *statement:*

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

If the *condition* is *true*, *statement1* is executed; if the condition is *false statement2* is executed. One or the other will be executed, but not both.

Example 4: Suppose `gpa` is a variable containing the grade point average of a student. Suppose the goal of a program is to let a student know if he/she made the Dean's list (the `gpa` must be 3.5 or above). Write an `if... else...` statement that prints out the appropriate message (either "`Congratulations—you made the Dean's List`" or "`Sorry you didn't make the Dean's List`").

Block statements

Several statements can be grouped together into a *block statement.* A block is delimited by braces : { … } A block statement can be used wherever a statement is called for by the Java syntax. For example, in an `if-else` statement, the `if` portion, or the `else` portion, or both, could be block statements.

```
if ( x == 3 )
   {
      statement1;
      statement2;
   }
else
   {
      statement3;
      statement4;
   }
```

Nested `if` statements

The statement executed as a result of an `if` statement or `else` clause could be another `if` statement. These are called *nested* `if` *statements*. An `else` clause is matched to the last *unmatched* `if` (no matter what the indentation implies). Braces can be used to specify the `if` statement to which an else clause belongs.

Example 5: What will the following code do if the user enters 7?

```
int n = scan.nextInt();
if (n > 0)
     if (n % 2 == 0)
          System.out.println (n);
else
     System.out.println (n + " is not positive");
```

Output:

Comparing Float Values

We also have to be careful when comparing two floating point values (`double`) for equality. You should rarely use the equality operator (==) when comparing two floats. 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 floats, you may want to use the following technique:

```
if (Math.abs(f1 - f2) < 0.00001)
   System.out.println ("Essentially equal.");
```

3.4 Comparing Strings

Comparing Strings

Remember that a character string in Java is an *object*. We cannot use the relational operators to compare strings. The `equals()` method, NOT the equals operator == can be called with strings to determine if two strings contain exactly the same characters in the same order. Case matters when using the `equals` method, therefore the string "Hi There" is not equal to "hi there". To determine if two strings are equal when you do not want the case to matter, use the `equalsIgnoreCase()` method.

```
String first = "Hi There";
String second = "hi there";

System.out.println(first.equals(second));
System.out.println(first.equalsIgnoreCase(second));
```

The String class also contains a method called `compareTo()` to determine if one string comes before another (based on the Unicode character set). Because comparing characters and strings is based on a character set, it is called a *lexicographic ordering.* This is not strictly alphabetical when uppercase and lowercase characters are mixed. For example, the string "Great" comes before the string "fantastic" because all of the uppercase letters come before all of the lowercase letters in Unicode. Also, short strings come before longer strings with the same prefix (lexicographically). Therefore "book" comes before "bookcase".

Example 6: Write an *expression* that evaluates to `true` if the value of the string variable `s1` is greater than the value of string variable `s2`.

Example 7: Use nested `if` statements to print out an appropriate message when the user enters a letter grade. "A" is Excellent, "B" is Good, "C" and "D" are Poor and "F" is Failure. Print a message if the user enters an invalid grade (not an A, B, C, D or F). Your program needs to work for both upper and lowercase user input.

3.5 The `while` Statement

Repetition statements

Repetition statements allow us to execute a statement multiple times. These are also called control structures that allow the computer to perform *iterative* tasks. Often they are referred to as *loops*. Like conditional statements, they are controlled by *boolean* expressions. You need to know the following control structures for the AP exam:

- the `while` *loop*
- the `for` *loop*

The programmer should choose the right kind of loop for the situation. When you can't determine how many times you want to execute the loop body, use a `while` statement. If you can determine how many times you want to execute the loop body, use a `for` statement.

`while` loop

The *while statement* has the following syntax:

```
while ( condition )
   statement;
```

Example 8: Determine the output of the following loop.

```
int i = 1, mult3 = 3;
while (mult3 < 20)
{
      System.out.print(mult3 + " ");
      i++;
      mult3 *= i;
}
```

Output:

Note that if the condition of a `while` statement is `false` initially, the statement is never executed. Therefore, the body of a while loop will execute zero or more times. The body of a `while` loop eventually must make the condition false. If not, it is an *infinite loop*, which will execute until the user interrupts the program. This is a common *logical* error.

Example 9: Determine the output of the following loop.

```
int power2 = 1;
while (power2 != 20)
{
      System.out.println(power2);
      power2 *= 2;
}
```

Output:

Example 10: The code below is supposed to print the integers from 10 to 1 backwards. What is wrong with it? (Hint: there are two problems!) Correct the code so it does the right thing.

```
int count = 10;
while (count >= 0)
{
   System.out.println(count);
   count = count + 1;
}
```

Sentinel value

Occasionally you may wish to process data input until a certain condition is met. You can specify a special value that signals the end of the input (for example "`n`" or `-1`). When the user enters the *sentinel* value, your program knows there are no more values to be entered.

Example 11: Complete the following code for averaging student test scores. Use a `while` loop and continue asking for test scores (integer values) until the user enters a `-1` to quit.

```
int total = 0, numGrades = 0, testScore = 0;
double average = 0.0;
boolean done = false;

Scanner scan = new Scanner(System.in);
System.out.print ("Enter student grade.  Enter -1 to Quit. ");
```

3.6 The `for` Statement

`for` statement

A `for` loop executes a known number of times. The `for` *statement* has the following syntax:

```
for ( initialization ; condition ; increment )
   statement;
```

Like a `while` loop, the condition of a `for` statement is tested prior to executing the loop body. Therefore, the body of a `for` loop will execute zero or more times. It is well suited for executing a loop a specific number of times that can be determined in advance.

Example 12: What is the output for the following `for` loop?

```
for (int k = 20; k >= 15; k--)
      System.out.print (k + " ");
```

Output:

Example 13: Given an `int` variable `n` that has been initialized to a positive value and in addition, `int` variables `k` and `total` that have already been declared, use a `for` loop to compute the sum of the cubes of the first `n` whole numbers, and store this value in `total`. Use no other variables other than `n`, `k`, and `total`.

A `for` loop is functionally equivalent to the following `while` loop structure:

```
initialization;
while ( condition )
{
   statement;
   increment;
}
```

Example 14: Write an equivalent `while` loop for the `for` loop above.

Nested Loops

Similar to nested `if` statements, loops can be nested as well. That is, the body of a loop can contain another loop. Each time through the outer loop, the inner loop goes through its full set of iterations.

Example 15: Determine the output for the two loops nested in an outer loop:

```
for (int i = 1; i<= 6; i++)
{
      for (int j = 1; j <= i; j++)
            System.out.print("+");
      for (int j = 1; j<= 6 – i; j++)
            System.out.print("*");
      System.out.println();
}
```

Output:

***Important Note:** Variables created in a `for` loop or inside a block statement {…} are deleted from memory immediately after the `for` loop or block statements finishes execution.

3.7 Program Development

Program Development

The creation of software involves four basic activities:

- establishing the requirements
- creating a design
- implementing the code
- testing the implementation

The development process is much more involved than this, but these are the four basic development activities:

- **Requirements**
  Software requirements specify the tasks a program must accomplish (**what** to do, not how to do it).

- **Design**
  A *software design* specifies **how** a program will accomplish its requirements. A design includes one or more *algorithms* to accomplish its goal. An *algorithm* is a step-by-step process for solving a problem. An algorithm may be expressed in *pseudocode*, which is code-like, but does not necessarily follow any specific syntax. In object-oriented development, the design establishes the *classes*, *objects*, *methods* and *data* that are required.

- **Implementation**
  Implementation is the process of translating a design into source code. Most novice programmers think that writing code is the heart of software development, but actually it should be the **least creative** step. Almost all important decisions are made during requirements and design stages. Implementation should focus on *coding detils*, including *style guidelines* and *documentation*.

- **Testing**
  A program should be executed multiple times with various *input* in an attempt to find errors. *Debubbing* is the process of discovering the causes of problems and fixing them. Programmers often think erroneously that there is "only one more bug" to fix. Tests should consider design details as well as overall requirements.