* Understanding java operators:
  + Java operator is a special symbol that can be applied to a set of variables, values or literals.
  + Three types of operators are available in Java: unary, binary, and ternary
  + If 2 operators have the same precedence, then java guarantees elft-to-right evaluation
  + Order of operator precedence:
    - Post- unary operators: expression++, expression—
    - Pre-unary operators: ++expression, --expression
    - -,+,~,!
    - \*./,%
    - +,-
    - <<,??,???
    - <.>.<=,>=,instance of
    - ==,!=
    - &,^,|
    - &&,||
    - Boolean expression ? expression1: expres-sion2
    - Assignment operators
* Working with Binary Arithmetic Operators
  + Most common operators in the java language
* Arithmetic operators: +,-,\*,/,%++,--
  + Parenthesis always takes precedence
  + PEMDMAS (parenthesis, exponents, multiplication, division, modulus, addition, subtraction)
* Numeric promotion
  + Numeric promotion rules
    - If two values have different data types, Java will automatically promote one of the values to the larger of the two data types
    - If one of the values s integral and the other is floating-point, java will automatically promote the integral value to the floating point values data type
    - Smaller data types, namely byte, short, and char are first promoted to int any time they are used with a java binary arithmetic operator, even if neither of the operands is int.
    - After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands
  + Floating point literally are assumed to be a double unless postfixed with an f, as in 2.1f
* Working with unary operators:
  + A unary operator is one that requires exactly one operand, or variable, to function.
  + They usually increase/decrease numeric values by one or negate Boolean values
  + Examples of unary operators: +,-,++,--,!
  + You cannot apply a negation operator – to a Boolean expression
  + You can’t apply ! to a numeric expression
* Using additional binary operators:
  + Int x = 1;
* Casting primitive types
  + Int x = (int)1.0;
  + Short y = (short)1921222; //stored as 20678
  + Int z = (int)9f;
  + Long t = 192301398193810323L;
* Compound assignment operators:
  + +=,-=
  + Compound operators can save us from having to explicitly cast a value.
  + Long x = 10, int y = 5; y\*=x; y will be casted to a long and apply the multiplication of two long values and then cast the result to an int
* Relational operators:
  + <,<=,>,>=
  + Relational instance of operator: a instanceof b – true if the reference that a points to is an instance of a class, subclass, or class that implements a particular interface, as named in b
* Logical operators
  + &,|,^ can be applied to both numeric and Boolean data types
  + When they are applied to Boolean data types, they are referred to as logical operators.
  + When they are applied to numeric data types, they are referred to as bitwise operators, as they preform bitwise comparison of the bits that compose the number.
  + X & Y (And): true only if both operands are true
  + X | Y (Inclusive or): false if both operands are false
  + X ^ Y (Exclusive or): true if the operands are different
  + && short-circuit operator
  + || short-circuit operator
  + &&,|| are identical to logical operators except that right-hand side of the expression may never be evaluated if the final result can be determined by the left hand side of the expression.
    - Example: x = true || (y<4);
  + A more common example of where short-circuit operators are used is checking for null objects before performing an operation, such as this
    - If (x!=null && x.getValue()<5){//do something}
      * If x was null, then the short-circuit prevents a null pointer exception from ever being thrown, since evaluation of x.getValue() is never reached
      * However, if && was replaced with & then it would throw a null pointer exception
* Equality operators:
  + There is a difference between two objects are the same and two objects are equivalent
  + == or != operators are used in 1 of three scenarios:
    - Comparing two numeric primitive types. If the numeric values are of different data types, the values are automatically promoted as previously described
      * 5 == 5.00 returns true since the left side is promoted to a double
    - Comparing two Boolean values
    - Comparing two objects, including null and string values.
    - For object comparison, the equality operator is applied to the references to the objects, not the objects they point to. Two reference are equal, if and only if they point to the same object or both point to null.
* Understanding Java statements (Java control flow statements):
  + Control flow statements break up the flow of execution by using decision making, looping, and branching, allowing the application to selectively execute particular segments of code.
* The if-ten-else statement:
  + Order is important when creating if-then-else statements.
* Ternary operators:
  + The conditional operator ? : otherwise known as the ternary operator is the only operator that takes three operands and is of the form
    - booleanExpression ? expression1 : expression2
  + System.out.println((Y<91)?9:”zebra”)
    - Compiler does not care because it converts everything to string
  + Int animal = (y<91> ? 9 : “Horse”; // does not compile
    - This does not compile because horse can not be assigned to an int
  + Only one of the right-hand expression of the ternary operator will be evaluated at runtime. In a manner similar to the short-circuit operators, if one of the two right-hand expressions in a ternary operator performs a side effect, then it may not be applied at runtime.
* The Switch statement:
  + It is a complex decision-making structure in which a single value is evaluated and flow is redirected to the first matching branch, known as a case statement
  + If no case statement is found that matches the value, an optional default statement will be called. If no such default option is available, the entire switch statement will be skipped.
  + Supported data types: byte and Byte, short and Short, char and Character, int and Integer, String, enum values
  + switch(variableToTest){
  + case constantExpression1:
  + //branch for case 1;
  + break;
  + case constantExpression2:
  + //branch for case 2;
  + break;
  + default:
  + //branch for default;
  + }
* Compile-time constant values
  + The value in each case statement must be compile-time constant values of the same data type as the switch value. This means you can use only literals, enum constants, or final constant variables of the same data type.
  + When there is no break statement, the code will. Jumping to the matching case and then execute all of the preceding case statements until it finds a break statement or finishes the structure.
  + Case statement values must also be a literal, enum constant, or final constant variable.
    - if a final variable that is passed to a method is used in the case statement then it will not compile even through it is final but it is not constant
* The While Statement:
  + Repetition control structure that has a termination condition implemented as a Boolean expression, that will continue as long as the expression evaluates to true
* The do-while statement:
  + Repetition control structure with a termination condition and statement, or block of statements.
  + Unlike a while loop, a do-while loop guarantees that the statement or block will be executed at least once.
  + do {
  + //body
  + }
  + while (booleanExpression);
* The for statement:
  + Variable declared within the for loop is only accessible within the foor loop.
  + For (intilizations;booleanExpressio;updateStatement)
  + Some examples of the for loop
    - Creating an infinite Loop that will compile:
      * For ( ; ; ){
        + Systemm.out.println(“hello”);
      * }
    - Adding multiple terms to the for statement
      * Int x = 0;
      * For (long y=0,z=4;x<5&&y<10;x++,y++){
        + System.out.println(y + “ “);
      * }
    - Redeclaring a variable in the initialization block
      * You can not re-declare a variable in the initialization block because compiler will fail on duplicate variables.
        + To get around this initialize both outside the for loop and simply set them to whatever you want in the initialization block
      * You can not use incompatible data types in the initialization block, variables in the initialization block must all be of the same type.
    - Using loop variables outside the loop:
      * You can not do that since the variables become out-of-scope once outside the loop
* The for-each statement:
  + For (datatype instance: collection)
    - The right-hand side must be a built -in Java array or an object whose class implements java.lang.Iterable, which includes most of the Java collections framework. The left-hand side of the for-each loop must include a declaration for an instance of a variable, whose type matches the type of a member of the array or collection in the right-hand side of the statement.
    - For-each loop hides access to the loop interior variable. If we wanted to print only the comma between names, we would convert the example into a standard for loop.
* Understanding Advanced flow control:
  + Nested Loops: covered them using code examples
* Adding optional labels:
  + A label is an optional pointer to the head of a statement that allows the application flow to jump to it or break from it. It is a single word that is proceeded bt a colon :
* The break statement:
  + A break statement transfers the flow of control out of the enclosing statement.
  + This also holdes true for break statements that appear inside a while, do-while, and for loops, as it will end the loop early.
  + The break statement can take an optional label parameter.
    - Without a label parameter, the break statement will terminate the nearest inner loop it is currently in the process of executing.
* The conture