Basic Java Review by Kevin Yonan

Java Types & Expressions

Intro

Programming languages are a combination of mathematical concepts from computer science and human language concepts such as syntax (grammar) and meaning (semantics). The key to understanding programming languages requires not just understanding the mathematical concepts but the grammar and semantics of the programming language.

Idea of Types:

Types are basically the collection/grouping of values, usually specified by a set of possible values, a set of allowed operations on these values, and/or a representation of these values for an abstract or concrete/physical machine.

In Java, there's a few primitive/basic type families and their types:

Integers (types that represent whole numbers):

- `byte` - 8-bit integer that represents the integer range of [-128 , 127]
- `char` - 16-bit integer for representing characters from text.
- `short` - 16-bit integer that represents the integer range of [−32,768 , 32,767]
- `int` - 32-bit integer that represents the integer range of [−2,147,483,648 , 2,147,483,647]
- `long` - 64-bit integer that represents the integer range of [−9,223,372,036,854,775,808 , 9,223,372,036,854,775,807]

Examples of integer constants/literals:

```
1       // decimal (base-10) constant
-1_000  // decimal (base-10) constant
0xA5    // hexadecimal (base-16) constant
077     // octal (base-8) constant
'A'     // character constant (uses single-quotes), same as integer value of 65.
```

Floating-Points (types that (imperfectly) represent real numbers):

- `float` - 32-bit floating point type.
- `double` - 64-bit floating point type.

Examples of floating-point literals/constants:

```
1.0          // real form constant of 1 (double type)
1.f          // real form constant of 1 (float type, float type requires 'f' suffix at end of constan
t)
0.5          // fractional form constant of 1/2 (double type)
1.e6         // exponential form constant of 1 million (double type)

0.00000149   // real form constant
1.49e-6      // same constant as previous but exponential form
```

Textual:

- `String` - represents text, this isn't a primitive but it is a basic type. It's also an `Object` type.

Example of textual literals/constants:

```
"this is a string"    // (text is enclosed with double-quotes)
"\t\n"                // escape characters representing a tab and newline (escape characters start wit
h back-slash)
""                    // empty string
```

list of escape characters and what they do:

```
\b -> Backspace
\f -> From Feed
\n -> Newline
\r -> Carriage Return
\t -> Horizontal Tab
\" -> Double Quote
\' -> Single Quote
\\ -> Backslash
```

Special:

- `boolean` - 8-bit value that represents two values: `true` and `false` .
- `void` - represents no return type for a method. Methods with `void` do not return any values after they complete their execution. [See the Methods section for more details]
- `Object` - represents the most basic object. Java is Object-Oriented and all user-defined classes use `Object` . The purpose of CSC205 aka Object-Oriented Programming with Java, is to teach you about objects.

Expressions

In programming languages and math, there's the concept of an *expression*. Expressions, in programming languages, are any line/ part of code that can compute a value. Here are some examples that use the basic data types we talked about:

`2 * 2` -> This expression is the multiplication of two integers, the type of the expression will thus be an integer type result. The result of course will be an `int`.

`2 * 2.0` -> this expression is the multiplication of an integer (left) and a floating-point value (right), the type of the expression will be a floating-point result and this is because mixing integers with floating-points, in an expression, are defined to result in a floating-point type. In this case, the `2` integer is converted to `2.0` then multiplied with the other `2.0` thus roughly yielding a floating-point type value of `4.0`.

`a >= 5` -> this expression is a comparison (greater-than-or-equal-to) between a variable and an integer, the expression's result will be a `boolean` type since the only posibility is whether `a` actually is greater-than-or-equal-to `5` or not.

Mixing data types where one type is larger in bits/bytes than the others usually results in an expression where the type is the largest sized data type. IE mixing `long` (64-bit) and `int` (32-bit) in an expression will give a result that is a `long` type.

`2 * 5 + b` -> this is a multi-expression (expression within expression). The expression (should) first compute the `2 * 5` and then compute the result of that value with `b`.

`(2 * 5) + b` -> same expression but nicely organized with parentheses.

`c` -> a variable by itself is considered a valid expression, the type of the expression is the type of the variable itself!

`"hi" + 1` -> a string added with a number (integer or floating-point) results in a string expression. The `1` is automatically converted from an `int` type into a `String` like "1" and then combined with "hi" to create `"hi1"`. This kind of automatic conversion is called **implicit conversion**.

`i + a.m()` -> expression where we have mathematic addition between a variable and a method call. This expression is only valid if `i` and the return value of the method `m` have compatible types. Typing rules, such as mixing integers and floating-point, still apply.

Operators

In Java, expressions work on a basis of operators and how they're defined for various types.

For Java, there's different categories of operators:

- Arithmetic operators
- Bitwise operators
- Comparison operators
- Logical operators
- Assignment operators
- Special operators

Arithmetic Operators (example):

- `+` - addition (`x + y`)

- `-` - subtraction (`x - y`)

- `-` - negate (`-x`)

- `+` - int promotion (`+x`) [promotes an integer expression's type to an `int` type]

- `*` - multiplication (`x * y`)

- `/` - division (`x / y`)

- `%` - modulo/integer division-remainder (`x % y`) [gives integer remainder]

  ```
  10 / 3 == 3
  10 % 3 == 1. Why?
  What's a multiple of 3 closest to 10? Answer: 9.
  What's 10 - 9? Answer: 1, thus the modulo of 10 with 3 is 1.

  26 % 9 == 8. Multiple of 9 closest to 26 is 18. 26 - 18 == 8.

  13 % 3 == 1, multiple of 3 closest to 13 is 12, so 13-12 == 1.
  9 % 3 == 0. multiple of 3 closest to 9 is 9, so 9 - 9 == 0.
  a % b == 0, if 'a' is a multiple of 'b'.
  ```

- `++` - increment: increases a numeric value by 1 (`++x`) [pre: increases value first in expression] (x++) [post: increases value after expression]

```
int a = 1;
System.out.println(a++);  // prints 1. 'a' is now 2.
System.out.println(++a);  // prints 3. 'a' is now 3.
```

- `--` - decrement: decreases a numeric value by 1 (`--x`) [pre: decreases value first in expression] (x–) [post: decreases value after expression]

```
int a = 1;
System.out.println(a--);  // prints  1. 'a' is now  0.
System.out.println(--a);  // prints -1. 'a' is now -1.
```

Bitwise Operators (example) [gives an integer type result]

- `&` - bitwise AND (`x & y`) [performs bitwise AND on all bits of an integer, in parallel]

```
             |---------------|
truth table: | A | B | A & B |
             |---------------|
             | 0 | 0 |   0   |
             |---------------|
             | 0 | 1 |   0   |
             |---------------|
             | 1 | 0 |   0   |
             |---------------|
             | 1 | 1 |   1   |
             |---------------|


6 & 11 == 0110 (decimal 6) & 1011 (decimal 11):


     0110
&    1011
---------
     0010 == 2
```

- `|` - bitwise OR (`x | y`)

```
             |---------------|
truth table: | A | B | A | B |
             |---------------|
             | 0 | 0 |   0   |
```

```
             |---------------|
             | 0 | 1 |   1   |
             |---------------|
             | 1 | 0 |   1   |
             |---------------|
             | 1 | 1 |   1   |
             |---------------|

5 | 3 == 0101 (decimal 5) | 0011 (decimal 3):

     0101
|    0011
---------
     0111 == 7
```

- `^` - bitwise XOR (`x ^ y`)

```
             |---------------|
truth table: | A | B | A ^ B |
             |---------------|
             | 0 | 0 |   0   |
             |---------------|
             | 0 | 1 |   1   |
             |---------------|
             | 1 | 0 |   1   |
             |---------------|
             | 1 | 1 |   0   |
             |---------------|

5 ^ 3 == 0101 (decimal 5) ^ 0011 (decimal 3):

     0101
^    0011
---------
     0110 == 8

a ^ a == 0
```

- `~` - bitwise NOT/complement (`~x`)

```
              |--------|
truth table:  | A | ~A |
              |--------|
              | 0 |  1 |
              |--------|
              | 1 |  0 |
              |--------|

~5 == ~0101 (decimal 5):

~    0101
---------
     1010 == 10
```

Bit Shifting

Only allowed on integer types, bit shifting let's you accomplish doing multiples of powers-of-two by shifting the bits of an integer type either to the left or to the right.

- `<<` - logical/arithmetic Left Bit Shift (`x << y`)

```
a << n == a*(2^n)

Example #1: 1 << 4 == 1*(2^4) == 2^4 == 2*2*2*2 == 16
Example #2: 3 << 3 == 3*(2^3) == 3*8 == 24
Example #2: 10 << 4 == 10*(2^4) == 10*16 == 160
```

- `>>` - arithmetic Right Bit Shift (`x >> y`)
- `>>>` - logical Right Bit Shift (`x >>> y`)

```
a >> n == a / (2^n)
Example: -4 >> 1 == -4 / (2^1) == -4 / 2 == -2


How '>>>' works:
sign bit == 2^(bit size of integer type - 1)
Example: 2^7 == 128 -> in binary: 1000 0000
                                  ^ - 7th bit index

non-sign bits == sign bit - 1
```

```
Example: 2^7 - 1 == 127 -> in binary: 0111 1111

all ones == sign bit + non-sign bits
Example: 2^7 + (2^7 - 1) == 1000 0000 + 0111 1111 == 1111 1111 (binary) or 0xff (hexadecimal)

No matter how many bits the integer makes up, all binary ones means it's all hexadecimal Fs.

a >>> n == (a & all ones) / (2^n)

Example (using `int` so 32-bit): 1 >>> 4 == (1 & 0xffff_ffff) / (2^4) == 1 / 16 == 0
Example: 4 >>> 1 == (4 & 0xffff_ffff) / (2^1) == 4 / 2 == 2

if 'a' is negative, we need to get its hex value, to start we think of its bits as opposite, all Fs/all ones is '-1'.
Assuming 8-bit integers:
-1 (binary 1111 1111) - 1 (binary 0000 0001) == -2 (binary 1111 1110)
So to take a number like -5 and convert it to hex, we get how it looks as a positive number: '0000 0101'
and then transform it like the '~' bitwise NOT operation: '1111 1010' (decimal -6) and then add 1 to it! (1111 1011) which is 0xFB!

Example: -4 >>> 1:
-4 -> take '4' and then flip everything: (0000 0000 0000 0000 0000 0000 0000 0100) -> (1111 1111 1111 1111 1111 1111 1111 1011)
Add one: (1111 1111 1111 1111 1111 1111 1111 1011) -> (1111 1111 1111 1111 1111 1111 1111 1100) which is in -4 in binary, 0xffff_fffc in hex.

Move the entire bit-set down by 1 and the value is the result of '-4 >>> 1':
(1111 1111 1111 1111 1111 1111 1111 1100) -> (0111 1111 1111 1111 1111 1111 1111 1110) which is 0x7ffffffe in hexadecimal

Identity Rule with shifting any direction with 0.
a << 0 == a >> 0 == a >>> 0 == a
```

Tricks using Bitwise Operators:

`a & a == a` . Identity Law

`a & 1 == 1` if `a` is odd, 0 if even.

`a & (a-1) == 0` if `a` is a power of 2.

`a & power-of-2-minus-1 == a % power-of-two` -> Example: `a % 32 == a & 31`.

`x = a ^ b ^ x` -> same as doing:

```
if( x==a )
    x = b;
else if( x==b )
    x = a;
```

Getting a specific bit position: `1 << bit position`
Example getting the 2nd bit position for an integer: `1 << 1 == 2`.

Setting the bit position `n` on an integer: `i |= (1 << n)`

Checking the bit position `n` on an integer: `i & (1 << n) > 0`

Toggling the bit position `n` on an integer: `i ^= (1 << n)`

Clearing/Removing the bit position `n` from an integer: `i &= ~(1 << n)`

Manipulating specific bit positions allow us to use an integer with N bits as N boolean values in one variable! Bit positions like these are also called **bit flags**. You can also do the same operations on multiple bit-flags at once:

Setting multiple bit positions on an integer: `i |= ( (1 << A) | ... | (1 << Z) )`

Checking multiple bit positions on an integer: `i & ( (1 << A) | ... | (1 << Z) ) > 0`

Toggling multiple bit positions on an integer: `i ^= ( (1 << A) | ... | (1 << Z) )`

Clearing/Removing multiple bit positions from an integer: `i &= ~( (1 << A) | ... | (1 << Z) )`

Examples:

```
int i = 0;

...;
```

```
i |= (2 | 4);    // set 2nd and 3rd bit at the same time.
i ^= (4 | 8);    // toggle 3rd and 4th bit.
if( (i & (2 | 8)) > 0 ) {    // test if 2nd and 4th bit are turned on.
    ...;
}


...;


i &= ~(-1);      // clear all active bits.
// recall that -1 is all 1s, so flipping all 1s makes it all 0s.
```

Comparison Operators (example) [gives a boolean type result]

- `>` - Greater-Than (`x > y`)
- `>=` - Greater-Than-Or-Equal-To (`x >= y`)
- `<` - Less-Than (`x < y`)
- `<=` - Less-Than-Or-Equal-To (`x <= y`)
- `==` - Equal-To (`x == y`)
- `!=` - Not-Equal-To (`x != y`)
- `instanceof` - Instance of “class” (`x instanceof y`) [used to check if an object is from a specific class]

Logical Operators (example) [gives a boolean type result]

- `&&` - logical AND (`x && y`)
- `||` - logical OR (`x || y`)
- `!` - logical NOT (`!x`)

Assignment Operators (example)

- `=` - Assignment/Copy (`x = y`)
- `+=` - Compound Addition (`x += y`) [same as: (`x = x + y`)]
- `-=` - Compound Subtraction (`x -= y`) [same as: (`x = x - y`)]
- `*=` - Compound Multiplication (`x *= y`) [same as: (`x = x * y`)]
- `/=` - Compound Division (`x /= y`) [same as: (`x = x / y`)]
- `%=` - Compound Modulo (`x %= y`) [same as: (`x = x % y`)]
- `&=` - Compound Bitwise-AND (`x &= y`) [same as: (`x = x & y`)]
- `|=` - Compound Bitwise-OR (`x |= y`) [same as: (`x = x | y`)]
- `^=` - Compound Bitwise-XOR (`x ^= y`) [same as: (`x = x ^ y`)]
- `<<=` - Compound Left Bit Shift (`x <<= y`) [same as: (`x = x << y`)]
- `>>=` - Compound Arithmetic Right Bit Shift (`x >>= y`) [same as: (`x = x >> y`)]
- `>>>=` - Compound Logical Right Bit Shift (`x >>>= y`) [same as: (`x = x >>> y`)]

Special Operators (example)

- `.` - Field/Member Access (`x.y`)

Operator Precedence

Similar to how math has PEMDAS/BODMAS to remember the precedence of each expression in math, Java and other programming languages also have their own defined operator precedence. Given that expressions can have inner expressions, it's important to know what operators and their operands are first processed your Java program runs.

The following table represents, from **highest** priority to **lowest** priority, what operators go first.

```
Level   Operator   Description          Associativity
------------------------------------------------------
16        (a)      parentheses          left-to-right
         a[b]      array access
          new      object creation
          a.b      member access
         a::b      method reference

        a(b,c,..)  method call
       a.b(c,d,..)
------------------------------------------------------
15        a++      unary post-increment  left-to-right
          a--      unary post-decrement
------------------------------------------------------
14        +a       unary plus           right-to-left
          -a       unary minus
          !a       unary logical NOT
          ~a       unary bitwise NOT
          ++a      unary pre-increment
          --a      unary pre-decrement
------------------------------------------------------
13     (type)a     type cast            right-to-left
------------------------------------------------------
12      * / %      multiplicative       left-to-right
------------------------------------------------------
11       + -       additive             left-to-right
          +        string concatenation
------------------------------------------------------
```

```
10     << >> >>>    shift                 left-to-right
-----------------------------------------------------
9      < <= > >=    relational            left-to-right
       instanceof
-----------------------------------------------------
8        == !=      equality              left-to-right
-----------------------------------------------------
7         &         bitwise AND           left-to-right
-----------------------------------------------------
6         ^         bitwise XOR           left-to-right
-----------------------------------------------------
5         |         bitwise OR            left-to-right
-----------------------------------------------------
4        &&         logical AND           left-to-right
-----------------------------------------------------
3        ||         logical OR            left-to-right
-----------------------------------------------------
2        ?:         ternary               right-to-left
-----------------------------------------------------
1       = += -=     assignment            right-to-left
      *= /= %=
      &= ^= |=
     <<= >>= >>>=
-----------------------------------------------------
```

so if we're given an expression like: `2 + 5 * 9 ^ 3 & 24 - 2 << 3`.

The result is of an `int` type with the value `47`.

The way Java and its Runtime has processed the number is as:

```
2 + (5 * 9) ^ 3 & 24 - 2 << 3             // multiplication first.
(2 + (5 * 9)) ^ 3 & (24 - 2) << 3         // addition then subtraction done next.
(2 + (5 * 9)) ^ 3 & ((24 - 2) << 3)       // bit shift done next.
(2 + (5 * 9)) ^ (3 & ((24 - 2) << 3))     // the Bitwise-AND done next.
(2 + (5 * 9)) ^ (3 & ((24 - 2) << 3))     // then the Bitwise-XOR done last.
```

Methods

Methods are for organizing code/functionality and mapping it to a name where it allows a developer to break down the complexity of a system into smaller, more manageable chunks.

Method Structure:

```
<zero-or-more modifiers> <return-type> <method name>(<comma-separated parameters>) <block statement>
```

Best Example when you start Java I is:

```
public static void main(String[] args) {
    ...;
}
```

the `public static void main(String[] args)` part, this is called the **method header or method signature**.

Referring back to the method structure, the method header is divided into 4 parts:
`public static` - modifiers.
`void` - return type of the method.
`main` - method name.
`(String[] args)` - method parameter(s). Methods can have zero, one, or as many parameters as needed.

The method header, when it also has the block statement with it [see Block Statement in the Statements section], the entire method is called the Method Definition or Method Declaration.

Basic Modifiers that you can use on methods:

```
public protected private static final
```

- `static` -> means the method is only tied to the class itself, not to any object of a class.
- `final` -> means the method cannot be overriden by subclasses (CSC205 will teach you what a subclass is).

The other modifiers are for controlling the visibility of methods (and class members).
Here's a table that explains the modifiers how much visibility they allow.

```
✔: accessible
❌: not accessible
------------------------------------------------------------------------------------------
            |  Class  |  Package  |  Subclass(same pkg)  |  Subclass(diff pkg)  |  World  |
------------------------------------------------------------------------------------------
```

```
public      |  ✔  |  ✔  |      ✔       |      ✔      |  ✔  |
-----------------------------------------------------------------------------------------
protected   |  ✔  |  ✔  |      ✔       |      ✔      |  ❌  |
-----------------------------------------------------------------------------------------
no modifier |  ✔  |  ✔  |      ✔       |      ❌      |  ❌  |
-----------------------------------------------------------------------------------------
private     |  ✔  |  ❌  |      ❌       |      ❌      |  ❌  |
-----------------------------------------------------------------------------------------
```

Parameters & Arguments

With methods, it's important to understand the definition and distinction of **parameters** and **arguments**.

A **Parameter** is a special variable used to refer to one of the pieces of data provided as input to the method.

On the other hand, an **Argument** is a value that's given as input to a method when the method is *invoked* or *called*. By extension, a parameter itself can be used as an argument to methods given that parameters are variables.

Parameter & Argument Example:

```
                  // vvvvvvvvvvvv - method name in definition.
public static double calcSalesTax(double price) {
                               // ^^^^^^^^^^^^ - 'price' is a parameter.
    return price * 0.06;  // using 'price' parameter in an expression!
}
...
public static void main(String[] args) {
                                           // vvvvvvvvvvvv - method name in method call expression.
    System.out.println("sales tax of $10: " + calcSalesTax(10.0));
                                                        // ^^^^^ - '10.0' is an argument.
                                                        // When control flow goes to 'calcSalesTax',
                                                        // 'price' will have the value of '10.0'.
}
```

Java Statements

Statements are any part/line of code that performs a basic computing action such as loading/storing data, managing control flow, or other things. Typically, statements can use expressions as part of their construct. Let's look at a few statements. Many statements also allow statements for them, allow for more intricate, complex code to accomplish more complex tasks.

Block Statement:

The most basic statement that begins using curly brackets and contains statements between said curly brackets.

```
{
    <statement1>;
    ...
    <statementN>;
}
```

Many other statements and constructs use the block statement as it's the only statement that allows multiple inner statements.

Expression Statement:

Most common statement, usually method calls or data manipulation of variables, basically a statement that is solely an expression. Usually, expression statements start using a variable or method name.

```
<expression>;
```

Examples:

```
i += 10;    // data manipulation - adding 10 onto 'i'.
a.m();      // method call.
a[n] += 4;  // data manipulation - adding 4 onto the index 'n' of array 'a'.
```

Variable Declarations & Initializations:

Variable Declaration Statement:

For basic data types or Objects: `<type-name> <variable name>`

For Arrays: `<type-name>[] <variable name>`

Example:

```
int   a;
int[] b;
```

Variable Initialization Statement:

For basic data types: `<variable name> = <expression that matches the type of the variable>`

For Arrays: `<variable name> = new <type of the variable>[]{<comma-separated expressions that matches the type of the variable>}`

For Objects: `<variable name> = new <type of the variable>(<comma-separated values for constructor>)`

Variable Declaration & Initialization Together:

For basic data types:

```
<type-name> <variable name> = <expression that matches the type-name>;
```

For Arrays:

```
<type-name>[] <variable name> = new <type-name>[<integer expression>]{<comma-separated expressions that match the type of the type-name>};
```

For Objects:

```
<type-name> <variable name> = new <type-name>(<comma-separated values for constructor>);
```

Example:

```
int a = 5;
int[] b = new int[3]{5, 7, 100};
MyClass g = new MyClass();
```

If you just want the array to start empty, omit/leave out the `{}` part:

```
<type-name>[] <variable name> = new <type-name>[<integer expression>];
```

Example:

```
double[] nums = new double[a * 3];
```

If-Statement:

Most basic control flow statement, runs code based on a boolean expression called a *condition*.

```
if (<boolean expression>) <statement>
```

Example:

```
if( a > 5 ) {
    System.out.println("a > 5");
}
```

If an `else` portion exists, it will run if the condition is false.

```
if (<boolean expression>) <statement>
else <statement>
```

Example:

```
if( (b & 1) != 0 ) {
    System.out.println("b is odd");
} else {
    System.out.println("b is even");
}
```

If an `else if` portion exists, it will run if the previous if-statement condition fails, running down as a cascade.

```
if (<boolean expression>) <statement>
else if (<boolean expression>) <statement>
```

Example:

```
if( c >= 100 && c <= 200 ) {
    System.out.println("100 <= c <= 200");
} else if( c > 200 ) {
    System.out.println("c > 200");
}
```

If both `else if` and `else` portions exist, it will work as a cascade and the `else` part will only run if the `if` and all the `else` `if` s have failed.

```
if (<boolean expression>) <statement>
else if (<boolean expression>) <statement>
...
else <statement>
```

Example:

```
if( grade >= 90 ) {
    System.out.println("grade: A");
} else if( grade >= 80 ) {
    System.out.println("grade: B");
} else if( grade >= 70 ) {
    System.out.println("grade: C");
} else if( grade >= 60 ) {
    System.out.println("grade: D");
} else {
    System.out.println("grade: F");
}
```

Ternary Expression:

If you ever come across a situation where you have to initialize a variable to a value that's based on an existing condition, rather than making a variable and then using an if-statement, you can alternatively use a ternary expression aka an inline if-expression.

```
<boolean expression> ? <expression if true> : <expression if false>
```

Example:

```
int i = (a > 5)? a * 5 : a + 5;
...
int x = ...;
System.out.println("is x greater than 7?: " + (x > 7? "yes" : "no"));
```

Since Ternary Expressions are themselves expressions, that means you can *nest* them but be careful doing this because it can hurt readability [ability for programmers to read and understand what the code is doing and why]. A good tip is to use parentheses to make it a bit clearer but still tread carefully.

:

```
int x_comparison = (x < 0)? -1 : ((x > 0)? 1 : 0);
```

Switch-Statement:

More advanced, compact form of the if-statement.

Most useful if you have to compare a singular item to alot of different values.

You can have as many cases as you need, as long as there’s no duplicate cases.

The `default` case is optional.

If you don’t have a `break` separating each case, the code control-flow will then fall through into the code of the next case!

`break` pretty much stops the control flow from going into the next case’s code.

`default` doesn’t need a `break` since `default` case has to always be the ending case.

```
switch (<integer or string expression>) {
    case <integer or string constant expression>: {
        <statement>
        break;
    }
    default:
        <statement>
}
```

Example:

```
switch( title ) {
    case "King": {
        System.out.println("Welcome your majesty!");
        break;
    }
    case "Queen": {
        System.out.println("Welcome your highness!");
        break;
    }
    case "Princess": {
        System.out.println("Greetings my princess!");
        break;
    }
    case "Prince": {
```

```
            System.out.println("Hello my prince!");
            break;
        }
        default: {
            System.out.println("Hello your grace.");
        }
}
```

Loop Statements:

For loops, they will run a statement until the boolean expression, called a *controller* or also called a *condition*, is `false` .

While Loops:

```
while (<boolean expression>) <statement>
```

Do-While Loops:

```
do <statement> while (<boolean expression>);
```

For Loops:

For loops are complicated, but more compact.

There are 3 parts to a for-loop.

First part is variable declaration or initialization. Typically called the `init` portion.

Second part is the condition as a boolean expression, this part controls the loop.

Third part is the post expression which runs after the statement body of the for-loop has finished executing.

All 3 parts are optional

```
for ( <variable declaration/initialization> ; <boolean expression> ; <post expression> ) <statement>
```

A for-loop is equivalent to a broken up while-loop:

```
<variable declaration/initialization>;
while (<boolean expression>) {
    <statement>
    <post expression>;
}
```

For Arrays and other collection objects [special objects that hold multiple data in a similar manner to arrays], there is a special for-loop syntax that helps simplify iterating/traversing these kinds of objects:

```
for (<variable declaration that matches type of array/collection> : <array/collection typed expression
>) <statement>
```

Example:

```java
int[] nums = new int[size];
fillNums(nums);
for( int num : nums ) {
    System.out.println(num);
}
```

Which is pretty much equivalent to:

```
for( int i=0; i < <container length value>; i++ ) {
    <variable declaration that matches type of array/collection>;
    <statement>
}
```

Using our example from previous:

```java
int[] nums = new int[array_size];
fillNums(nums);
for( int i=0; i < nums.length; i++ ) {
    int num = nums[i];
    System.out.println(num);
}
```

Loop Controllers: Flow Statements

- `continue` -> skips the current iteration of the loop and goes to the next.
- `break` -> completely stops the loop.

NOTE: be careful using a switch statement in a loop because the `break` needed for the cases will NOT stop the loop.
If you have to stop the loop some how and you want a switch statement within the loop, use another variable, preferrably a `boolean` type, to be able to kill the loop. Your homework is figuring out how to do that!

Return Statement:

Used in methods to return to its previous calling location and, if the method doesn't have a `void` return type, gives back a value.

```
return <expression that matches the return type of the method>
```

Example:

```
public static int findIndexOfValueInArray(int[] numbers, int value) {
    for( int i=0; i < numbers.length; i++ ) {
        if( numbers[i] == value ) {
            return i;
        }
    }
    return -1; // error value.
}
```

NOTE:: the **`int`** part of the method, that's the return type of the method, *and* the return value of i which is an `int` -based integer expression.

If the method has a `void` return type, then no expression is given to the `return` :

```
public static void enterVote(int age, int candidate_number, int[] votes) {
    if( age < REQUIRED_AGE ) {
        System.out.println("sorry you're not old enough to vote in this election.");
        return;
    }
    votes[candidateNumToIndex(candidate_number)]++;
}

public static void sortVotes(int[] votes) {
    if( votes.length <= 1 ) {
        return;
    }

    for( int i=0; i < votes.length; i++ ) {
        for( int j=0; j < votes.length; j++ ) {
            if(i != j && votes[j] < votes[i]) {
                int temp = a[i];
```

```
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

Try-Catch Statement:

The try-catch statement is for doing exception/error handling, allowing the program to “try” a block of code and “catch” any exceptions (errors) that may occur during execution. It’s important to be able to handle errors as they happen to make the program more robust and continue running. Without handling exceptions/errors, a program would cease every time whether unexpectedly or the user is assaulted with an error message they might not understand.

At its most basic:

```
try <block statement>
catch (<exception type> <variable name>) <block statement>
```

You can have as many catch portions as you need to handle as many errors as needed.

```
try <block statement>
catch (<exception type> <variable name>) <block statement>
...
catch (<exception type> <variable name>) <block statement>
```

Also as an option, you can use a `finally` portion that will **always** execute regardless whether an error happened or not:

```
try <block statement>
catch (<exception type> <variable name>) <block statement>...
finally <block statement>
```

Example without try-catch:

```
int[] myNumbers = {1, 2, 3};
System.out.println(myNumbers[10]); // error!
```

Console Message:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 3
```

Example *with* try-catch:

```
int[] myNumbers = {1, 2, 3};
try {
    System.out.println(myNumbers[10]);
} catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Error :: ****" + e + "**** | try a different index.");
}
```

Not exactly recommended but if it's necessary to catch ALL errors with one catch block, you can use the `Exception` type. It's not recommended because sometimes different errors need to be handled differently but, if you're planning to handle all errors the same way, then simply using `Exception` is fine. Special errors I'd recommend handling specially are `NullPointerException` (trying to use an object that's null) and `ArithmeticException` (such as dividing by 0).

```
int[] myNumbers = {1, 2, 3};
try {
    System.out.println(myNumbers[10]);
} catch(Exception e) {
    System.out.println("Error :: ****" + e + "****");
}
```

Putting Logic into Code: Applied Programming

Using Methods to Refactor and Organize Code

Often if you have repetitive code, it's best to abstract that code into a method.

Using letters in place of each line of code, what pattern of code do you notice in this abstract example?:

```
a; b; c;
x;
a; b; c;
y;
a; b; c;
z;
```

Hopefully you notice that the code doing `a`, `b`, and `c` repeat three times. In such a case, this repetition can be better organized by having a method that has a single set of `a`, `b`, and `c` and we can just repeat these lines of code by calling the method:

```
method doThing() {
    a; b; c;
}
```

Thus the code in the abstract example can now be simplified into:

```
doThing();
x;
doThing();
y;
doThing();
z;
```

Example of repetitive code:

```java
import java.util.Scanner;

public class QuadraticFormula {
    public static void main(String[] args) {
        System.out.println("Welcome to the Quadratic Formula Solver | Ax^2 + Bx + C = 0");
        Scanner s = new Scanner(System.in);

        System.out.println("Enter A: ");
        double a = s.nextDouble();

        System.out.println("Enter B: ");
        double b = s.nextDouble();

        System.out.println("Enter C: ");
        double c = s.nextDouble();

        double x1 = (-b + Math.sqrt( (b*b) - (4*a*c) )) / (2*a);
        double x2 = (-b - Math.sqrt( (b*b) - (4*a*c) )) / (2*a);

        System.out.println("root x1: " + x1);
        System.out.println("root x2: " + x2);
```

```
    }
}
```

Let's rewrite our earlier example in the abstract pseudocode to better highlight what kind of patterns we can find.

let's define `System.out.println` as `a`.

let's define `double <var name> = s.nextDouble()` as `b`.

```
public class QuadraticFormula {
    public static void main(String[] args) {
        a("Welcome to the Quadratic Formula Solver | Ax^2 + Bx + C = 0");
        Scanner s = new Scanner(System.in);

        a("Enter A: ");
        b;

        a("Enter B: ");
        b;

        a("Enter C: ");
        b;

        double x1 = (-b + Math.sqrt( (b*b) - (4*a*c) )) / (2*a);
        double x2 = (-b - Math.sqrt( (b*b) - (4*a*c) )) / (2*a);

        a("root x1: " + x1);
        a("root x2: " + x2);
    }
}
```

There is a common repetition between `a` and `b` in the code. We can wrap `a` and `b` into a method and have it return something. What would the parameters be though? Well the parameters would be everything that our `a` and `b` uses in itself.

So earlier, we defined `a` and `b` as:

`System.out.println` as `a`.

`double <var name> = s.nextDouble()` as `b`.

In the context of where `a` is used in the repetitive pattern, all it does is print a string so we'll need a `String` parameter.

In the context where `b` is used in the repetitive pattern, `b` itself uses the `Scanner` object `s` so that means we'll need a `Scanner` parameter as well.

That just about covers the parameters but what about the return type? For the return type, we analyze the pattern to see what data is usually set or given from the pattern. In `b`, we see that the `Scanner` reads a `double` from input and puts it into a variable of a different name each time the abstract line of code `b` is executed. We can take advantage of this by returning the `double` that was read from input.

```
public static double getDoubleFromInput(String msg, Scanner s) {
    System.out.println(msg);      // code 'a'.
    double d = s.nextDouble();    // code 'b'.
    return d;
}
```

since we're returning the `double` variable that has the `double` we read from input, we can further simplify the method by just returning the input reading directly:

```
public static double getDoubleFromInput(String msg, Scanner s) {
    System.out.println(msg);  // code 'a'.
    return s.nextDouble();    // still technically code 'b'.
}
```

This is also a good time to talk about method naming. Like with variables, you should give methods a name that reflects their purpose and usage. I chose the name `getDoubleFromInput` because that's mostly what the method does. Although it also prints a string variable named as `msg` to the user, the purpose of it is to alert the user to give an input, so the name for it clearly reflects its purpose and usage.

With the method, we can replace the abstract lines of code we defined as `a` and `b` with the method call and supply the values in their place as arguments to the method call:

before:

```
System.out.println("Enter A: ");
double a = s.nextDouble();
```

after:

```
double a = getDoubleFromInput("Enter A: ", s);
```

Thus, when we apply the method to the other repeating lines, we get a more simplified version of our previous code:

```
import java.util.Scanner;

public class QuadraticFormula {
    public static double getDoubleFromInput(String msg, Scanner s) {
        System.out.println(msg);
        return s.nextDouble();
    }
    public static void main(String[] args) {
        System.out.println("Welcome to the Quadratic Formula Solver | Ax^2 + Bx + C = 0");
        Scanner s = new Scanner(System.in);

        double a = getDoubleFromInput("Enter A: ", s);
        double b = getDoubleFromInput("Enter B: ", s);
        double c = getDoubleFromInput("Enter C: ", s);

        double x1 = (-b + Math.sqrt( (b*b) - (4*a*c) )) / (2*a);
        double x2 = (-b - Math.sqrt( (b*b) - (4*a*c) )) / (2*a);

        System.out.println("root x1: " + x1);
        System.out.println("root x2: " + x2);
    }
}
```

By refactoring our code into methods, we avoid repetition, make our code more organized, and make it easier to modify in the future. If the input process changes, we only need to update our method instead of multiple lines in the main code.

We can also go further with the code by simplifying the quadratic formula computation.
Let's now define `Q1` as the line of code `(-b + Math.sqrt( (b*b) - (4*a*c) )) / (2*a)`
and define `Q2` as the line of code `(-b - Math.sqrt( (b*b) - (4*a*c) )) / (2*a)` .

Thing is, we can also abstract `Q1` and `Q2` themselves.
Let's define `X` as `Math.sqrt((b*b) - (4*a*c))` and `Y` as `(2*a)` .
So now `Q1` and `Q2` are redefined as `(-b + X) / Y` and `(-b - X) / Y` respectively.
In a sense, we've made `Q1` and `Q2` into pseudomethods since they both take `X` and `Y` as pseudoparameters:

```
Q1(X, Y)
Q2(X, Y)
```

Given the abstract code definitions and the subdefinitions, we can surmise that the parameters we'd need for our method is `a`, `b`, and `c` all as `double` type as they're the only values being used. Since `Math.sqrt` is public method, we don't need to get it as a parameter somehow. One thing we do need to keep in mind is that `Math.sqrt` returns a `double` value.

Another thing is what about our return type? We have a `Q1` and a `Q2`, which means we need to somehow return two values but how do we return two `double` values at the same time? Since `Q1` and `Q2` are both expressions of type `double`, that means we can enclose the two results in a `double` array! This will allow us to return both X roots of the quadratic formula computation without having to do a separate method for either the addition or subtraction part with `-b`.

So our overall method now looks like:

```
public static double[] computeQuadratic(double a, double b, double c) {
    double sqrt_val = Math.sqrt( (b*b) - (4*a*c) );   // X
    double two_a = 2*a;                               // Y
    double x1 = (-b + sqrt_val) / two_a;              // Q1(X, Y)
    double x2 = (-b - sqrt_val) / two_a;              // Q2(X, Y)
    return new double[]{ x1, x2 };
}
```

So our example code now looks like:

```
import java.util.Scanner;

public class QuadraticFormula {
    public static double getDoubleFromInput(String msg, Scanner s) {
        System.out.println(msg);
        return s.nextDouble();
    }
    public static double[] computeQuadratic(double a, double b, double c) {
        double sqrt_val = Math.sqrt((b*b) - (4*a*c));
        double two_a = 2*a;
        double x1 = (-b + sqrt_val) / two_a;
        double x2 = (-b - sqrt_val) / two_a;
        return new double[]{ x1, x2 };
    }
    public static void main(String[] args) {
        System.out.println("Welcome to the Quadratic Formula Solver | Ax^2 + Bx + C = 0");
        Scanner s = new Scanner(System.in);
```

```
        double a = getDoubleFromInput("Enter A: ", s);
        double b = getDoubleFromInput("Enter B: ", s);
        double c = getDoubleFromInput("Enter C: ", s);
        double[] Xs = computeQuadratic(a, b, c);
        System.out.printf("roots x1: %f, x2: %f\n", Xs[0], Xs[1]);
    }
}
```

Our code is now the most organized it can be. We've effectively organized the code to retrieve a specific number of inputs after alerting the user to do so, perform calculations of the retrieved input, and then display the computation back to the user. Try finding repetitive code in a different example, and think about how you might refactor it using methods.

Separation of Concerns

```
What it is: The idea that different parts of a program should handle different aspects of functionalit
y. For example, UI code should be separate from the logic or data manipulation.
Why it’s important: Helps students understand how to organize code into different modules or classes b
ased on their roles. Improves readability, testability, and maintainability.
Example: Separate business logic from presentation in a basic MVC (Model-View-Controller) pattern.
```

Code Refactoring Techniques

```
Concept: Refactoring is the process of restructuring existing code without changing its external behav
ior to improve readability, structure, or performance.
Why It’s Important: Clean code is easier to maintain and extend, and refactoring helps avoid technical
debt.
Application: Teach students common refactoring patterns such as extracting classes, simplifying condit
ionals, or breaking down large methods. Use examples like:
    Extract Method (splitting large methods into smaller, reusable ones)
    Inline Variable (removing unnecessary local variables)
    Replace Magic Numbers (with named constants)
```