APCS 2020 Review

Welcome to the APCS review, here you will find content summaries of each unit along with code examples to help you build a conceptual and practical understanding of the content. Units on Exam:

1. Primitive Types
2. Using Objects
3. Boolean Logic and Conditionals
4. Iteration
    - For Loops
    - While Loops
    - Searching and Sorting
5. Writing Classes
    - Writing Methods
6. Arrays
7. ArrayLists

FRQ Examples

Units not on 2020 exam (still important):

1. 2D Arrays
2. Inheritance
3. Recursion

Minimal Fundamentals

In this section we will go over the very basics of Java to give you a quick refresher. You can probably skip over this.

Variable declaration is: Type name;

```
int num;
```

Variable initialization is: name = value;

```
num = 5;
```

Both can be combined with: Type name = value;

```
int num = 5;
```

Basic math is performed with the following operators: + - * /

```
x + 5;
```

Modulo (%) is the remainder in division, ex:

```
5 % 3 == 2;
```

A shorthand for changing a variables value:

```
x = x + 5;
x += 5;
x++; // increments by 1
```

Output is done like this:

```
System.out.print("print");
System.out.println("println"); // print() is used for printing text without moving to the n
ext line
System.out.println("println2"); // println() prints the text, and "presses enter"
```

The cell below runs some java code to demonstrate these features:

In [1]:

```
// Declaration
int num;

// Initialization
num = 5;

System.out.println(num);

// Both
int x = 4;
```

```
5
```

In [2]:

```
// Addition example
System.out.println(x + 5); // 4 + 5 = 9

// Modulo (%) is the remainder in division, ex:
System.out.println(x % 3); // 4/3 has a remainder of 1
```

```
9
1
```

In [3]:

```
// A shorthand for changing a variables value:
x = x + 5;
System.out.println(x); // 4 + 5 = 9
x += 5;
System.out.println(x); // 9 + 5 = 14
x++;
System.out.println(x); // 14 + 1 = 15
x--;
System.out.println(x); // 15 - 1 = 14
```

```
9
14
15
14
```

In [4]:

```
// Printing:
System.out.print("print");
System.out.println("println");
System.out.println("println2");
// print() is used for printing text without moving to the next line
// println() prints the text, and "presses enter"
```

```
printprintln
println2
```

Primitive Types

The types to be seen on the exam are:

- `int` (whole numbers)
- `double` (decimal numbers)
- `boolean` (true or false values)
- `char` (single characters)
- not primitive but important: `String` (text, an array of characters)

Tips:

- Integer division will truncate the result, as seen below. For precision beyond whole numbers, use doubles

- Integer division will truncate the result, as seen below. For precision beyond whole numbers, use doubles
- Generally, one uses `==` to compare two variables. With strings you have to use `.equals()` to check equality
- Boolean logic is easiest to evaluate inside out
- When converting between types, you must cast to the desired type

Boolean comparison table:

a	operator	b	result
true	&&	true	true
false	&&	false	false
true	||	true	true
false	||	false	false
true	&&	false	false
true	||	false	true

In [5]:

```
// Integer usage:
int a = 5;
int b = 2;

System.out.println("--Integers--");
System.out.println(a + " + " + b + " = " + (a + b));
System.out.println(a + " - " + b + " = " + (a - b));
System.out.println(a + " * " + b + " = " + (a * b));
System.out.println(a + " / " + b + " = " + (a / b));
```

```
--Integers--
5 + 2 = 7
5 - 2 = 3
5 * 2 = 10
5 / 2 = 2
```

In [6]:

```
// Double usage:
double a = 20;
double b = 0.5;

System.out.println("--Doubles--");
System.out.println(a + " + " + b + " = " + (a + b));
System.out.println(a + " - " + b + " = " + (a - b));
System.out.println(a + " * " + b + " = " + (a * b));
System.out.println(a + " / " + b + " = " + (a / b));
```

```
--Doubles--
20.0 + 0.5 = 20.5
20.0 - 0.5 = 19.5
20.0 * 0.5 = 10.0
20.0 / 0.5 = 40.0
```

In [7]:

```
// Boolean usage:
boolean a = true;
boolean b = false;

System.out.println("--Booleans--");
System.out.println("Negation:");
System.out.println("!" + a + " = " + !a);
System.out.println("!" + b + " = " + !b);
```

```
--Booleans--
Negation:
!true = false
!false = true
```

In [8]:

```
System.out.println("Boolean Operators:");
System.out.println(a + "  && " + a + " = " + (a && a));
System.out.println(b + " && " + b + " = " + (b && b));
```

```
Boolean Operators:
true  && true = true
false && false = false
```

In [9]:

```
System.out.println(a + "  || " + a + " = " + (a || a));
System.out.println(b + " || " + b + " = " + (b || b));
```

```
true  || true = true
false || false = false
```

In [10]:

```
System.out.println(a + " && " + b + " = " + (a && b));
System.out.println(a + " || " + b + " = " + (a || b));
```

```
true && false = false
true || false = true
```

In [11]:

```
// Basic String usage
String name = "Will";
System.out.println("Name: " + name);
```

```
Name: Will
```

In [12]:

```
// Casting
int x = 10;
double newx = (double) (x);

double y = 2.5;
int newy = (int) (y);

System.out.println(x + " -> " + newx);
System.out.println(y + " -> " + newy);
```

```
10 -> 10.0
2.5 -> 2
```

Using Objects

Here we'll review some commonly used objects such as the `Math`, `Scanner`, `String` classes.

- `Math`: has methods for more complex mathematical tasks.
- `Scanner`: used to get input from the console or files
- `String`: array of chars for storing text

Important Math methods

- `sqrt(x)`: returns the square root of x
- `pow(base, x)`: returns base to the power of x
- `log(x)`: returns the natural log of x
- `log10(x)`: returns the log base 10 of x
- `random()`: returns a random number between 0 and 1, but cannot be 1. Can be scaled through multiplication, addition, and subtraction to fit any range you like

subtraction to fit any range you like

In [13]:

```
// Common Math methods:
System.out.println("sqrt(9): " + Math.sqrt(9));
System.out.println("3^2 or pow(3, 2): " + Math.pow(3, 2));
System.out.println("ln(2.71828) -> log(): " + Math.log(2.71828));
System.out.println("log10(1000) -> log10(): " + Math.log10(1000));
```

```
sqrt(9): 3.0
3^2 or pow(3, 2): 9.0
ln(2.71828) -> log(): 0.999999327347282
log10(1000) -> log10(): 3.0
```

In [14]:

```
System.out.println("random in [0.0, 1.0) -> random(): " + Math.random());
System.out.println("random in [0.0, 5.0) -> random() * 5: " + Math.random() * 5);
System.out.println("random in [-1.0, 1.0) -> random() * 2 - 1.0: " + (Math.random() * 2 - 1.0));
```

```
random in [0.0, 1.0) -> random(): 0.2534280364111593
random in [0.0, 5.0) -> random() * 5: 2.9365475533907266
random in [-1.0, 1.0) -> random() * 2 - 1.0: -0.05742384815488166
```

In [15]:

```
// Common Math constants:
System.out.println("Math.PI: " + Math.PI);
System.out.println("Math.E: " + Math.E);
```

```
Math.PI: 3.141592653589793
Math.E: 2.718281828459045
```

Scanner Usage:

To use a scanner, you must create a Scanner object and give it `System.in` for user input. This is shown below. The scanner has many methods that do the same thing for different data types, below I will put an `X` where the name of a data type can go. Supported data types with scanners include int, double, boolean, and String. For String, X is empty. For any other type, just fill in the X with what you need.

Important methods:

- `hasNextX()` : ignoring any whitespace, returns a boolean whether there is a next token and it is type `X`
- `nextX()` : if `hasNextX()` is true, returns the next `X` from its input

In [16]:

```
// Scanner usage:
// 1. Declare and initialize for console input
// Type name = new Type();
Scanner console = new Scanner(System.in);

// Show prompt
System.out.print("Enter a number: ");
// calling console.nextInt() will get int from user
// console.next() gives a word
// console.nextDouble() gives double
int x = console.nextInt();
System.out.println("Number you entered: " + x);
```

```
Enter a number:

Number you entered: 4
```

In [17]:

```
// Scanner can also be used on a string
String text = "apcs will be the easiest exam";
Scanner textReader = new Scanner(text);

// This loop will run over and over until textReader runs out of words to give
while (textReader.hasNext()) {
    System.out.print(textReader.next() + "_");
}
```

```
apcs_will_be_the_easiest_exam_
```

Important String methods

- `length()` : returns the length of the string. ex: `"abcde".length()` will be `5`
- `substring(int a)` : returns the string starting at index `a`. ex: `"01234".substring(2)` will be `"234"`
- `substring(int a, int b)` : returns the string starting at index `a`, ending at `b` exclusive. ex: `"01234".substring(1, 3)` will be `"12"`
- `charAt(int a)` : returns the char at index `a`. ex: `"abcde".charAt(2)` is `'c'`
- `indexOf(String target)` : returns index of `target` in the string, and `-1` if not found. ex: `"abc april".indexOf("bc")` is `1`
- `split(String delimiter)` : splits the string up by the delimeter and returns the array of results. ex: `"word1 word2".split(" ")` is `["word1", "word2"]`

  Note: you can add strings or characters with +. ex: `"abc" + "def"` will give `"abcdef"`

In [18]:

```
// String usage
String text = "apcs will be the easiest exam";

// Adding indexes for reference:
System.out.print("index:");
for (int i = 0; i < text.length(); i++) System.out.print(i % 10);
System.out.println("\ntext: " + text);

System.out.println("text.length(): " + text.length());
System.out.println("text.substring(11): " + text.substring(11));
System.out.println("text.substring(0, 11): " + text.substring(0, 11));
System.out.println("text.charAt(13): " + text.charAt(13));
System.out.println("text.indexOf(\"easiest\"): " + text.indexOf("easiest"));
System.out.println("text.split(\" \"): " + Arrays.toString(text.split(" ")));
System.out.println("text.split(\"e\"): " + Arrays.toString(text.split("e")));
```

```
index:01234567890123456789012345678
text: apcs will be the easiest exam
text.length(): 29
text.substring(11): e the easiest exam
text.substring(0, 11): apcs will b
text.charAt(13): t
text.indexOf("easiest"): 17
text.split(" "): [apcs, will, be, the, easiest, exam]
text.split("e"): [apcs will b,  th,  , asi, st , xam]
```

Booleans and Conditionals

In this section we will review boolean usage and conditional statements

Conditional structures:

- `if (this) {A}` : if `this` is true, then run `A`
- `if (this) {A} else {B}` : same as first, but if `this` is false, run `B`. Only A or B can be run, not both
- `if (this) {A} else if (that) {B} else {C}` same as second but if `that` is true, then run `B`. Only one of A, B, or C can be run.

In [19]:

```
int x = 5;
int y = 6;

if (x == y) {
    System.out.println("x == y");
} else if (x + 1 == y) {
    System.out.println("x + 1 == y");
} else {
    System.out.println("x != y and x + 1 != y");
}
```

```
x + 1 == y
```

In [20]:

```
if (x == y - 1) {
    System.out.println("x == y - 1");
}
if (x + 1 == y) {
    System.out.println("x + 1 == y");
}
```

```
x == y - 1
x + 1 == y
```

In [21]:

```
boolean a = true;
boolean b = false;
if (a && b) {
    System.out.println("a and b are true");
} else if (a != b) {
    System.out.println("a is not b");
}

if (a || b) {
    System.out.println("a or b is true");
}
```

```
a is not b
a or b is true
```

Iteration

Here we'll review iterative structures such as the for loop and while loop

Tips:

- Use for loop for a known number of iterations (ex: going from 1 to 10)
- Use while loop for an unknown number of iterations (ex: going until you find a number divisible by 24 and 49)

For Loop Syntax

```
for (init; condition; statement) {
    code;
}
```

Upon reaching the for loop, `init` is run. Then, if `condition` is true, `code` will be run. Next, `statement` is run. `condition` is checked again and the process cycles.

`init` -> `condition` -> `code` -> `statement` -> `condition`

In [22]:

```
for (int i = 0; i < 5; i++) {
    System.out.print(i);
}
```

```
01234
```

In [23]:

```
for (int i = 10; i > 0; i--) {
    System.out.print(i + " ");
}
```

```
10 9 8 7 6 5 4 3 2 1
```

In [24]:

```
for (int i = 1; i < 10; i *= 2) {
    System.out.print(i + " ");
}
```

```
1 2 4 8
```

While Loop Syntax

```
while (condition) {
    code;
}
```

Upon reaching the loop, if `condition` is true, `code` will be run. Then `condition` is checked again, and it keeps running.

In [25]:

```
int numPeople = 10;
int numPizzas = 0;

System.out.println("People: " + numPeople);
System.out.println("Pizzas: " + numPizzas);
```

```
People: 10
Pizzas: 0
```

In [26]:

```
while (numPizzas < numPeople) {
    numPizzas += 1;
}

System.out.println("People: " + numPeople);
System.out.println("Pizzas: " + numPizzas);
```

```
People: 10
Pizzas: 10
```

In [27]:

```
while (numPeople > 0) {
    numPeople -= 1;
}

System.out.println("People: " + numPeople);
System.out.println("Pizzas: " + numPizzas);
```

```
People: 0
Pizzas: 10
```

Searching and Sorting

Searching: Iterative Search

With regards to searching an unsorted list, or even a small/medium unsorted one, the best way is a simple loop through each element. Note the amount of time this method takes is linearly proportional to the length of the list. An array with 10x more elements will, on average, take 10x longer.

Note: If you are unfamiliar with the fundamentals of arrays and ArrayLists, read about them here

In [28]:

```
Integer[] searchMe = {1, 53, 8, 42, 59, 33, 2, 591, 52};
Integer target = 33;
int index = 0;

// Note the order of the conditions: if you reached the end, don't bother to check the array or it
'll be out of bounds
while (index < searchMe.length && !searchMe[index].equals(target)) {
    index += 1;
}
// Either the index is equal to the length, or we found the element
if (index == searchMe.length) {
    System.out.println("Target not found");
} else {
    System.out.println("Target found at index " + index);
}
```

```
Target found at index 5
```

Sorting: Selection Sort

Selection sort is one of the main sorting algorithms in APCS, and it is straightforward when analyzed step by step.

1. Split the list into sorted and unsorted, and initially all elements are unsorted
2. Find the minimum in the unsorted part
3. Swap it with the leftmost element in the unsorted part
4. That element is now sorted, go to step 2 until you reach the second to last
5. The final one is sorted because the rest are

Split, Minimum, Swap, Repeat

In practice, this is pretty easy to implement. Keep in mind we have the for loop wrapped around steps 2 and 3 to go from the start to the end of the list. Additionally, if you are wondering about the method, read the section on writing methods

In [29]:

```
public void selectionSort(int[] arr) {
    // Step 1: Split into sorted/unsorted
    // We use sortedIndex as our marker
    for (int sortedIndex = 0; sortedIndex < arr.length - 1; sortedIndex++) {
        // Step 2: Find minimum in unsorted section
        int minIndex = sortedIndex;
        for (int i = sortedIndex; i < arr.length; i++) {
            if (arr[i] < arr[minIndex]) {
                minIndex = i;
            }
        }
        // i is now the index of the minimum in the unsorted section

        // Step 3: Swap the first unsorted with the minimum
        int temp = arr[sortedIndex];
        arr[sortedIndex] = arr[minIndex];
        arr[minIndex] = temp;
        // Add a print statement to show the list at each iteration
        System.out.println(Arrays.toString(arr));
    }
}
```

In [30]:

```
int[] sortMe = {4, 17, 84, 55};
System.out.println("Unsorted: " + Arrays.toString(sortMe));
selectionSort(sortMe);
```

```
selectionSort(sortMe);
System.out.println("Sorted: " + Arrays.toString(sortMe));

int[] sortMe = {39, 24, 12, 0, 59};
System.out.println("\nUnsorted: " + Arrays.toString(sortMe));
selectionSort(sortMe);
System.out.println("Sorted: " + Arrays.toString(sortMe));
```

```
Unsorted: [4, 17, 84, 55]
[4, 17, 84, 55]
[4, 17, 84, 55]
[4, 17, 55, 84]
Sorted: [4, 17, 55, 84]

Unsorted: [39, 24, 12, 0, 59]
[0, 24, 12, 39, 59]
[0, 12, 24, 39, 59]
[0, 12, 24, 39, 59]
[0, 12, 24, 39, 59]
Sorted: [0, 12, 24, 39, 59]
```

Sorting: Insertion Sort

Insertion sort is also straightforward, and somewhat more efficient than selection sort. It is very similar, here are the steps:

1. Mark the first element as sorted
2. Look at the first unsorted element, `X`
3. Search the sorted section right to left until you find an element, `Y`, smaller than `X`
4. Insert `X` to the right of `Y`, and move the barrier between unsorted and sorted
5. Go to step 2 until the whole list is sorted

Note that this is best implemented with an arraylist because of the insertion involved.

In [31]:

```
public void insertionSort(ArrayList<Integer> aList) {
    // Step 1 & 5
    for (int sortedIndex = 1; sortedIndex < aList.size(); sortedIndex++) {
        // Step 2
        // The element to be moved is at sortedIndex, its potential spot is newIndex
        int newIndex = sortedIndex;

        // Step 3
        while (newIndex > 0 && aList.get(newIndex - 1) > aList.get(sortedIndex)) {
            newIndex -= 1;
        }
        // Step 4
        // Now we are either at index 0 or the new position for the element to be moved
        aList.add(newIndex, aList.remove(sortedIndex));
    }
}
```

In [32]:

```
Integer[] sortArr = {-4, 63, 35, 3, 20, 44, -40};
ArrayList<Integer> sortAL = new ArrayList<Integer>(Arrays.asList(sortArr));
System.out.println("Unsorted: " + sortAL);
insertionSort(sortAL);
System.out.println("Sorted: " + sortAL);
```

```
Unsorted: [-4, 63, 35, 3, 20, 44, -40]
Sorted: [-40, -4, 3, 20, 35, 44, 63]
```

Writing Classes

This is by far one of the most intricate topics in APCS, so it is important to have a good intuition of the topic.

Before we get into classes, let's do a quick review of writing methods.

Writing Methods

Methods are a powerful tool in computer science, and are used to reduce redundancy (repetition) and bring structure to a program. The generic method looks like:

```
// Header
access (static) Type name(ParamType param, AnotherParamType anotherParam,...) {
    // Body
    code;
    code;
    return result;
}
```

Let's break this down:

- `access` : this is a keyword representing the level of access for the method. If public, then anyone can call the method. If private, it can only be called inside the class.
- `(static)` : this keyword can either be present or absent, and determines if the method is static (more on that below)
- `Type` : when writing a method, you must declare the type that it will return, whether it be `int` , `boolean` , or even an object. Additionally, if the method does not return anything, then `Type` must be void
- `name` : name of the method, concise and summarizes the function
- `ParamType` / `AnotherParamType` : if the method has parameters, you must list the type for each
- `param` / `anotherParam` : these are the names of the parameters. Keep in mind that when calling the method the names do not matter, only the order
- `code` : you can put any code you like inside the method, and it can access `param` and `anotherParam`
- `return result` : for a non-void method (returns something), you must have this statement in any possible path in your method. Additionally, `result` must be of the `Type` you declared in the header

Let's look at some methods:

`getRandom(int a, int b)`

This method utilizes `Math.random()` to generate an integer from a up to b. Do not worry about the implementation, focus on the method header.

- `public` tells us that anyone can call this method, not just other methods in the class
- `int` tells us the return type of this method; whenever we call this method we can be 100% sure that it will return an int
- `getRandom` is a descriptive, concise name
- `int a` and `int b` are the parameters for the method, notice that we have to explicitly define the types
- `return (int) (result)` is the final statement that will give the value `result` as an integer back to the caller of the method

In [33]:

```
// Returns a random number between a and b exclusive
public int getRandom(int a, int b) {
    double result = Math.random();
    int range = b - a;
    result *= range;
    result += a;
    return (int) (result);
}

for (int i = 0; i < 20; i++) {
    System.out.print(getRandom(1, 10) + " ");
}
```

```
9 7 2 8 1 5 1 6 8 8 5 5 2 5 3 7 9 6 7 9
```

`reverseArray(int[] arr)`

This method will take an array and reverse it. The important thing about this example is the usage of reference semantics. When a parameter of a method is an object (such as an array), the reference will be passed into the method, not a copy of the object. If I call `reverseArray(myArr)` , by design, it will modify `myArr` . This is why the method is void.

In [34]:

```
// Reverses an array of ints
public void reverseArray(int[] arr) {
    for (int i = 0; i < arr.length / 2; i++) {
        int temp = arr[i];
        arr[i] = arr[arr.length - 1 - i];
        arr[arr.length - 1 - i] = temp;
    }
    System.out.println("Reversed!");
}

int[] myArr = {1, 2, 3, 4, 5, 6, 7, 8, 9};
System.out.println(Arrays.toString(myArr));
reverseArray(myArr);
System.out.println(Arrays.toString(myArr));
reverseArray(myArr);
System.out.println(Arrays.toString(myArr));
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
Reversed!
[9, 8, 7, 6, 5, 4, 3, 2, 1]
Reversed!
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Class Definition

A class is a "template" for an object. For example, if `Car` is a class, then `myHondaAccord` is a `Car` object. An object (instance) is an example of a class. `lily` is an instance of the `Dog` class. Classes and objects are defined by 2 things, state and behavior. These are represented with fields (variables) and methods respectively. Take an example `Dog` class:

In [35]:

```
// For convenience, I will not make the instance fields private. Know that they 100% should be pri
vate.
// Class header
public class Dog {
    // Instance Fields
    String name;

    // Static Fields
    static int numLegs = 4;
    static String foodType = "meat";

    // Constructor
    public Dog(String name) {
        this.name = name;
    }

    // Instance method bark
    public void bark() {
        System.out.println("woof");
    }

    // Static method getFoodType
    public static String getFoodType() {
        return foodType;
    }

    public String toString() {
        return name + " (Dog)";
    }
}
```

Fields

The distinction between classes and objects allows us to define 2 types of fields and methods: static (shared) and instance (individual). A static field in the `Dog` class could be `hasTail`, a boolean with a value that is the same across all instances. An instance field would be `name`, which varies depending on which instance (dog) you are talking about. The purpose of static fields is general values such as constants across all instances. Static fields can be accessed with `Type.field` while instance fields are

accessed with `object.field`. Another thing to note is that there is only one value of a static field that is used by all instances, where instance fields are unique and private. Take a look at this example:

In [36]:

```
Dog lily = new Dog("Lily");

System.out.println(lily.name); // "Lily"
System.out.println(Dog.numLegs); // 4

System.out.println(lily.numLegs); // 4
System.out.println(Dog.name); // does not exist, will cause error
```

```
Lily
4
4
```

```
|   System.out.println(Dog.name); // does not exist, will cause error
non-static variable name cannot be referenced from a static context
```

Methods

We have the same thing with methods, where static methods are shared across instances and vice versa. Static methods are used for utility operations whereas instance methods are used when the method needs information from instance fields for an action specific to one instance. Here's an example:

In [37]:

```
Dog lily = new Dog("Lily");

System.out.println(Dog.getFoodType()); // "meat"
System.out.println(lily.getFoodType()); // "meat"

lily.bark(); // Lily barks
Dog.bark(); // this doesn't work, will cause error
```

```
meat
meat
woof
```

```
|   Dog.bark(); // this doesn't work, will cause error
non-static method bark() cannot be referenced from a static context
```

Constructors

Here is the constructor from the `Dog` class, let's take a closer look at it.

```
    public Dog(String name) {
        this.name = name;
    }
```

The important things here are the header and the instance field assignment. The header for any constructor will be `public Type(params)` where `Type` is the name of the class. This is an important thing to remember as it's different from other methods.

Next, look at the line in the constructor. Since we have a collision between the instance field `name` and the parameter `name`, how will the computer know which is which? We use the `this` keyword to solve this. The `this` keyword will always refer to this instance, so in the line in the constructor we are setting the instance field `this.name` to the parameter `name`'s value.

toString()

The `toString` method is an optional method included in classes that is used whenever an instance is referred to as a string. This is more straightforward than it sounds. Look at the `toString` for the `Dog` class:

```
    public String toString() {
        return name + " (Dog)";
```

```
    }
```

Whenever this method is called, it will return the `name` of the `Dog` it was called on plus some clarification as to the class. The toString method makes it much easier to display the state of the instance. We can see this in action with a simple comparison between the `Dog` toString and the default `Object` toString:

In [38]:

```
Dog lily = new Dog("Lily");
Object bad = new Object();

System.out.println(lily);
System.out.println(bad);
```

```
Lily (Dog)
java.lang.Object@1571597b
```

A proper example

This is a `Vector` class I wrote awhile ago, and I think it would be useful to read through the annotations and understand the design choices made.

In [39]:

```
public class Vector {
    // Instance fields
    private int x;
    private int y;

    // i and j basis vectors, static because constants
    public static final Vector i = new Vector(1, 0);
    public static final Vector j = new Vector(0, 1);

    // Constructor to take (x, y) and make new vector
    public Vector(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Constructor to duplicate another vector
    public Vector(Vector v) {
        this.x = v.getX();
        this.y = v.getY();
    }

    // Find the distance from this vector to the other:
    // d(v1, v2) = sqrt((dx)^2 + (dy)^2)
    public double getDistance(Vector v) {
        int dx = v.getX() - x;
        int dy = v.getY() - y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    // Add two vectors: V1 + V2 = [V1x + V2x, V1y + V2y]
    public Vector addVector(Vector v) {
        int newX = x + v.getX();
        int newY = y + v.getY();
        return new Vector(newX, newY);
    }

    // Scale a vector: V * k = [Vx * k, Vy * k]
    public Vector scale(double scalar) {
        int newX = (int) (x * scalar);
        int newY = (int) (y * scalar);
        return new Vector(newX, newY);
    }

    // Get the X value
    public int getX() { return x; }

    // Get the Y value
    public int getY() { return y; }
```

```
    // Show vector state
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

In [40]:

```
Vector V1 = new Vector(1, 2);
Vector V2 = new Vector(4, 6);
System.out.println("i: " + Vector.i);
System.out.println("j: " + Vector.j);
System.out.println();

System.out.println("V1: " + V1);
System.out.println("V2: " + V2);
System.out.println();

System.out.println("V1.getDistance(V2): " + V1.getDistance(V2));
System.out.println("V1.addVector(V2): " + V1.addVector(V2));
System.out.println("V1.scale(2): " + V1.scale(2));
```

```
i: (1, 0)
j: (0, 1)

V1: (1, 2)
V2: (4, 6)

V1.getDistance(V2): 5.0
V1.addVector(V2): (5, 8)
V1.scale(2): (2, 4)
```

Arrays

Arrays are an integral part of computer science. Their function is to hold a defined number of a certain type. You can have an array of ints, chars, Strings, Dogs, or anything you like. The most significant limitation of arrays is their static size, meaning they cannot grow to fit more elements. This has its pros and cons, but here we will address the syntax:

Declaration and Initialization

```
Type[] name = new Type[num];
```

`Type` is the class of items in the array, `name` is the variable to hold the reference to the array, and `num` is the number of items that the array can hold. You may notice that there is a `new` keyword in the intialization. This indicates that arrays are actually objects. So one thing to keep in mind is `name` does not refer to the array itself, but a reference. If you call a method with it like `copy(name)`, that method receives the reference, not the array. So any changes on will be seen by both sides.

In [41]:

```
// Initialize blank
int[] arr = new int[3];

// Initialize with elements:
int[] arr = {1, 2, 3};
```

Accessing and Modifying

```
// Accessing
Type a = name[i];

// Modifying
Type b = new Type();
name[i] = b;
```

This syntax is straightforward. The example shown takes the element of `name` at index `i` and puts it into `a` (the same type as the array). Looking at modification, this is intuitive. We create a new `Type` with reference `b`, and set `name`'s `i`th element to be `b`

array). Looking at modification, this is intuitive. We create a new `type` with reference `b`, and set `name`'s `i`th element to be `b`.
Here is an example:

In [42]:

```
System.out.println("arr[2]: " + arr[2]);
arr[2] = 100;
System.out.println("arr[2]: " + arr[2]);
```

```
arr[2]: 3
arr[2]: 100
```

Iteration with Arrays

For Loops

For loops make it very easy to iterate through an array. The general syntax is:

```
for (int i = 0; i < arr.length; i++) {
    code(arr[i]);
}
```

This loop will go through all the indexes from 0 to `arr.length - 1`, all the valid indexes. At each index, it will call `code(element)` for the element in `arr` at `i`.

Another type of iteration is a for-each loop, which makes it even easier to look at all the elements in an array. However, you cannot use a for-each loop if:

- you need access to the indexes (`i`)
- you need to modify the array

For-Each Loops

Here is the syntax for a for-each loop:

```
for (int x : arr) {
    code(x);
}
```

This loop will go through each `int` `x` in `arr` and call `code(x)`. This loop can make things much more convenient or inconvenient, so use your judgement to see which one fits the problem best. If in doubt, use the traditional for loop.

Let's look at some common examples for ideas you could implement.

In [43]:

```
// Make an array of names
String[] classNames = new String[5];
classNames[0] = "Marie";
classNames[1] = "Smith";
classNames[2] = "John";
classNames[3] = "Michael";
classNames[4] = "Arjun";

// We will use Arrays.toString to print the array, otherwise it will give us a reference to the ar
ray, not its elements
System.out.println(Arrays.toString(classNames));
```

```
[Marie, Smith, John, Michael, Arjun]
```

In [44]:

```
// Print every name
// Something to think about: How do you fix the extra "and"?
for (String name : classNames) {
    System.out.print(name + " and ");
}
```

```
Marie and Smith and John and Michael and Arjun and
```

In [45]:

```
// Print the names backwards
for (int i = classNames.length - 1; i >= 0; i--) {
    System.out.print(classNames[i] + " ");
}
```

```
Arjun Michael John Smith Marie
```

In [46]:

```
// Reverse the array itself
System.out.println(Arrays.toString(classNames));
for (int i = 0; i < classNames.length / 2; i++) {
    String tempName = classNames[i];
    classNames[i] = classNames[classNames.length - 1 - i];
    classNames[classNames.length - 1 - i] = tempName;
}
System.out.println(Arrays.toString(classNames));
```

```
[Marie, Smith, John, Michael, Arjun]
[Arjun, Michael, John, Smith, Marie]
```

In [47]:

```
// Add a clarification to all the names
System.out.println(Arrays.toString(classNames));
for (int i = 0; i < classNames.length - 1; i++) {
    classNames[i] += " (Student)";
}
System.out.println(Arrays.toString(classNames));
```

```
[Arjun, Michael, John, Smith, Marie]
[Arjun (Student), Michael (Student), John (Student), Smith (Student), Marie]
```

ArrayLists

Definitions

Array Lists are an incredibly powerful data structure that is covered in this class.

Array Lists are very similar to arrays in the respect that they are able to store multiple instances of the same primitive type and objects. However, the key difference is that their size changes automatically.

Think of this like a todo list your phone. You can keep adding tasks to your todo list and its size grows to accomodate the new item(s) added. Furthermore you can pass something simple like a title (e.g. Do homework) or add more details to your task (like how you can store primitives and objects). Regardless of the complexity of these tasks, however, they are all treated like objects (ArrayLists do not take in (int)'s but rather (Integer)'s).

Constructing an array list

In [48]:

```
ArrayList al = new ArrayList(); // creating old non-generic arraylist
ArrayList<String> al = new ArrayList<String>(); // creating new generic arraylist (preferred)
```

Methods

The following methods are in the java quick reference but here's a quick review of them.

`size()` : returns the number of elements in the list
`add(obj)` : adds an object to the end of the list and returns true

`add(obj)` : adds an object to the end of the list and returns true
`add(index, obj)` : adds an object to the list at a specified index
`get(index)` : gets an object at the specified index
`set(index, obj)` : sets an object at the specified index and returns object formerly in that position `remove(index)` : removes an object at the specified index, shifts objects after the removed one down a spot, and returns the object removed

Notes:

- get, set, and remove all return an object at a specified position. For example:

In [49]:

```
Integer[] testCases = {1, 1, 2, 2, 3, 4, 5};
ArrayList<Integer> nums = new ArrayList<Integer>();
```

In [50]:

```
// for each loop (works on arrays and arraylists) to add all values of testCases to nums
for (int num : testCases) {
    nums.add(num);
}
```

In [51]:

```
// for an arraylist named nums containing: [1, 2, 3, 4, 5]
System.out.println(nums.remove(0)); // removes item at 0 (1) and returned value
System.out.println(nums);
```

```
1
[1, 2, 2, 3, 4, 5]
```

- also remember that remove takes an object away from the list so you have to be careful whenever you iterate through the list with that method:

In [52]:

```
// for an arraylist named nums containing: [1, 2, 2, 3, 4, 5]
// this for loop is to remove all instances of 2 in nums
for (int i = 0; i < nums.size(); i++) {
    if (nums.get(i) == 2) {
        nums.remove(i);
    }
    System.out.println(nums);
}
```

```
[1, 2, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

The previous code will remove the 2 at index 1 and shift all the elements of the arraylist down. That means that the second 2 is now at index one. The i gets incremented and skips over that second 2.

See the following code examples for solutions:

In [53]:

```
// solutions for code ex
System.out.println("Solution - Iterating backwards:");
// iterating backwards
nums = new ArrayList<Integer>(Arrays.asList(testCases));
for (int i = nums.size() - 1; i >= 0; i--) {
  if (nums.get(i) == 2) {
      nums.remove(i);
  }
  System.out.println(nums);
```

```
}
```

```
Solution - Iterating backwards:
[1, 1, 2, 2, 3, 4, 5]
[1, 1, 2, 2, 3, 4, 5]
[1, 1, 2, 2, 3, 4, 5]
[1, 1, 2, 3, 4, 5]
[1, 1, 3, 4, 5]
[1, 1, 3, 4, 5]
[1, 1, 3, 4, 5]
```

In [54]:

```
System.out.println("Solution - While loop:");
// while loop
nums = new ArrayList<Integer>(Arrays.asList(testCases));
for (int i = 0; i < nums.size(); i++) {
  while (nums.get(i) == 2) {
      nums.remove(i);
  }
  System.out.println(nums);
}
```

```
Solution - While loop:
[1, 1, 2, 2, 3, 4, 5]
[1, 1, 2, 2, 3, 4, 5]
[1, 1, 3, 4, 5]
[1, 1, 3, 4, 5]
[1, 1, 3, 4, 5]
```

FRQ Examples

Here are some previous year's FRQ's done and explained

These explanations cover our thought processes as we did the FRQ (including mistakes) in hopes that you can implement some of our tactics with analyzing the problem, solving it, and dealing with any setbacks.

NOTE: Everyone has their own style of writing code and its best to use what's most comfortable for you. Just because it worked for us does not necessarily mean it is ideal for you. As a result, use these examples as a way to get insight on how we think through some of these FRQs and see how you can apply it to your own methods.

In [55]:

```
// 2019 FRQ 1

public class APCalendar {

    // part a

   public static int numberOfLeapYears(int year1, int year2) {
      int counter = 0;
      for (int i = year1; i < year2; i++) {
         if (isLeapYear(i)) {
            counter++;
         }
      }
      return counter;
   }

   public static int dayOfWeek(int month, int day, int year) {
      int startDay = firstDayOfYear(year);

      int numDay = dayOfYear(month, day, year);

      return (startDay + (numDay - 1)) % 7;
   }


   public static boolean isLeapYear(int year) {
      return year % 4 == 0;
   }
```

```
   private static int firstDayOfYear(int year) {
      return 0;

   }

   public static int dayOfYear(int month, int day, int year) {
      return 0;
   }

   public static void main(String[] args) {
      System.out.println(numberOfLeapYears(2003, 2005)); //2003, 2004, 2005 (should be one)
      System.out.println(numberOfLeapYears(2003, 2010)); //(should be two)


   }



}
```

Explanation

Part A: numberOfLeapYears

Question Overview

The requirements of the question are as follows:

- Count the number of leap years between year 1 and year 2
- Implement the isLeapYear method which returns if a specific year is a leap year or not

Thought Process:

isLeapYear is a boolean and therefore its implementation is probably with an if statement if we follow this path of using the if statement, then we we will most likely need to iterate through the range of years and count how many of those years are leap years

Implementation:

I personally jumped in and wrote a quick for loop that would implement my thought process:

```
    for (int i = year1; i < year2; i++) {
             if (isLeapYear(i)) {
                counter++;
             }
          }
```

And then I added in the code bits that would declare and return the counter variable.

I also added a dummy method isLeapYear() to quickly test if my code could compile. I also noticed that I could quickly test if my code worked so I added a short main method. **Do note that this may not work for this year's AP Test.**

Part B:

Question Overview

The requirements of the question are as follows:

- return an integer that represents the day of the week a day is on (0 for sunday, 1 for monday, etc.)
- Implement the firstDayOfYear method which returns which day of the week a year starts on
- Implement dayOfYear which returns n which represents the nth day of the year given the specified date

Thought Process:

It's asking about days of the week:

- Its asking to represent them from 0 - 6

- its asking to represent them from 0 - 6
- There are 7 days in a week Therefore, the question is more than likely going to use mod 7 in some way

Now what other information do we have?

1. we know which day of the week is the first day of the given year (which is in mod 7)
2. we know how many days are between the first day and our specified day of the year (dayOfYear - 1 to not overcount day 1)

What can we do with that?

if we put the 2nd piece of information in mod 7, we know which day of the week it falls on relative to day 1.

- eg if they were to be on the same day, it would end up being 0 mod 7

so if we simply add the 2 we could get the actual position in the week. $startDay + (numDay - 1) \% 7$

To verify this, I checked limiting cases (i.e. making startDay and numDay - 1 as large and as small as possible). I soon found a problem as it might go over 7 as the mod only acts on the second term (i.e. startDay = 6, numDay - 1 = 5 % 7, total = 11)

so to fix that we could do this instead: $(startDay + numDay - 1) \% 7$

Implementation:

lets implement the first method: `int startDay = firstDayOfYear(year);`

same thing for the second: `int numDay = dayOfYear(month, day, year);`

now return the math: `return (startDay + (numDay - 1)) % 7;`

I couldn't see a quick way to test this so I simply added dummy methods and made sure the code compiled.

In [56]:

```
// 2019 FRQ B
public class StepTracker {
   private int activeDays;
   private int totalSteps;
   private int minSteps;
   private int totalDays;

   public StepTracker(int minSteps) {
      this.minSteps = minSteps;
      this.activeDays = 0;
      this.totalSteps = 0;
      this.totalDays = 0;
   }

   public int activeDays() {
      return activeDays;
   }

   public void addDailySteps(int dailySteps) {
      if (dailySteps >= minSteps) {
         activeDays++;
      }

      totalSteps += dailySteps;
      totalDays++;

   }

   public double averageSteps() {
      if (totalDays == 0) {
         return 0;
      }
      return (double) totalSteps / totalDays;
   }

   public static void main(String[] args) {
      StepTracker tr = new StepTracker(10000);
      System.out.println(tr.activeDays());
      System.out.println(tr.averageSteps());
      tr.addDailySteps(9000);
      tr.addDailySteps(5000);
      System.out.println(tr.activeDays());
      System.out.println(tr.averageSteps());
```

```
        tr.addDailySteps(13000);
        System.out.println(tr.activeDays());
        System.out.println(tr.averageSteps());
    }


}
```

Explanation

Requirements:

- StepTracker: creates an instance of StepTracker and passes the minimum steps for an "active day"
- addDailySteps: adds how many steps taken in a day and if its over the minimum step threshold, make today an active day
- activeDays: return the number of active days
- averageSteps: returns the average number of steps per day

Process:

Based on the requirements, I created 3 instance variables to serve as placeholders to fulfill those requirements:

```
private int activeDays;
private double averageSteps;
private int minSteps;
```

Next I created the constructor to satisfy the minSteps requirement and set everything else to 0.

```
public StepTracker(int minSteps) {
      this.minSteps = minSteps;
      this.activeDays = 0;
      this.averageSteps = 0;
}
```

Now I implemented the addDailySteps method:

- I checked to see if it was greater than or equal to minSteps and if it was it would increment active days

```
    public void addDailySteps(int dailySteps) {
        if (dailySteps >= minSteps) {
           activeDays++;
        }
    }
```

  Then as I was trying to figure out how this could update average steps, I realized that with my current instance variables, it wouldn't be easy to figure out the average steps.

So then I split up averageSteps into totalDays and totalSteps, changed my constructor, and finished addDailySteps():

```
public void addDailySteps(int dailySteps) {
      if (dailySteps >= minSteps) {
         activeDays++;
      }

      totalSteps += dailySteps;
      totalDays++;

   }
```

From there, activeDays and averageSteps were basically trivial:

```
public int activeDays() {
      return activeDays;
}

public double averageSteps() {
```

```
        return (double) totalSteps / totalDays;
    }
```

Then I quickly took a few of the methods from the example to test the functionality of my class:

```
    public static void main(String[] args) {
        StepTracker tr = new StepTracker(10000);
        System.out.println(tr.activeDays());
        System.out.println(tr.averageSteps());
        tr.addDailySteps(9000);
        tr.addDailySteps(5000);
        System.out.println(tr.activeDays());
        System.out.println(tr.averageSteps());
        tr.addDailySteps(13000);
        System.out.println(tr.activeDays());
        System.out.println(tr.averageSteps());
    }
```

Interestingly, the second print returned NaN and I realized that before incrementing totalDays I would be trying to divide by zero.

So I updated averageSteps() and called it a day:

```
    public double averageSteps() {
        if (totalDays == 0) {
            return 0;
        }
        return (double) totalSteps / totalDays;
    }
```

In [57]:

```
import java.util.*;
public class Delimiters {

   private String openDel;
   private String closeDel;
   public Delimiters(String open, String close) {
      openDel = open;
      closeDel = close;
   }
   // part a
   public ArrayList<String> getDelimitersList(String[] tokens) {
      ArrayList<String> delList = new ArrayList<String>();
      for (String str : tokens) {
         if (str.equals(openDel) || str.equals(closeDel)) {
            delList.add(str);
         }
      }
      return delList;
   }


   // part b
   public boolean isBalanced(ArrayList<String> delimiters) {
      int open = 0;
      int closed = 0;
      for (String str : delimiters) {
         if (str.equals(openDel)) {
            open++;
         } else {
            closed++;
            if (closed > open) {
               return false;
            }
         }
      }
      return open == closed;
   }
}
```

Explanation

Part A:

Requirements:

- Get an array of Strings
- Add all instances of delimeters to an arraylist of strings and return it

Thought Process:

We can probably just iterate through the array and do some if to add the delimeters to the arraylist

Implementation:

Since we only care about the values of the array, we can use a for-each loop:

```
for (String str : tokens) {

}
```

Now we need the if statement:

```
if (str.equals(openDel) || str.equals(closeDel)) {

}
```

Now we need to add it to an arraylist that I forgot to declare at the start;

```
ArrayList<String> delList = new ArrayList<String>();
for (String str : tokens) {
    if (str.equals(openDel) || str.equals(closeDel)) {
        delList.add(str);
    }
}
```

Now we just add the return statement: `return delList;`

To check the code, I wrote the constructor and instance fields they had and compiled the code.

Part B:

Requirements:

- Make sure there is no point on the arraylist where there are more closed delimeters than open delimeters (when iterating through)
- Make sure that there are the same amount of open and closed delimeters

Process:

We are probably going to need 2 counters to be able to complete the requirements. From there we can iterate through the arraylist and check things as we go.

Implementation:

Create the 2 counters:

```
int open = 0;
int closed = 0;
```

Iterate through the for loop and update the counters:

```
for (String str : delimiters) {
    if (str.equals(openDel)) {
```

```
            open++;
        } else {
            closed++;
    }
```

Now we need to satisfy the first condition and since `closed` only changes value whenever it gets incremented, we can insert the check then.

```
    else {
        closed++;
        if (closed > open) {
            return false;
        }
    }
```

Now we just need to satisfy the second condition (remember boolean zen): `return open == closed;`