Certified Hood Study Material
Code credits by JM

Extra Materials:

1 | Click here
2 | Click here
3 | FRQ Scoring Guidelines

System.out.println(... args) → prints args then prints a new line (\n)

a. Can take no arguments to just print a new line

System.out.print(... args) → println without the new line at the end

Comments:

```
// Single Line Comment

/*
 * Block comments
 */

/**
 * JavaDocs style Block Comments
 * @param means the definition of a parameter
 * @return means the value returned
 */
```

Common Core Terms:
Boxed Expression (here either means .equals() or a wrapper class in this document)

Primitives :

Int:
-boxed expression : (java.lang.Integer)

a. Primarily used for generics, such defining what type of objects are going in an ArrayList (java.util.ArrayList)

-represents a whole number or a regular integer
-IS NOT FLOATING POINT
-defaults initialization to 0
-an int can be directly assigned to a double

`Integer(int value)`	Constructs a new `Integer` object that represents the specified `int` value
`Integer.MIN_VALUE`	The minimum value represented by an `int` or `Integer`
`Integer.MAX_VALUE`	The maximum value represented by an `int` or `Integer`
`int intValue()`	Returns the value of this `Integer` as an `int`

-can hold in the range of -2,147,483,648 to 2,147,483,647

a. If the value of this integer is exceeded, an IntegerOverflowException will be thrown and the entered value will be become minuscule (negative)
b. Integer.MIN_VALUE & Integer.MAX_VALUE holds these ranges

```
Integer t = new Integer(100);
int x = 100;
```

Double:

-boxed expression : (java.lang.Double)

a. Primarily used for generics, such as defining what type of objects are going in an Arraylist (java.util.ArrayList)

-represents a FLOATING POINT EXPRESSION

-default initialization to 0.0

`Double(double value)`	Constructs a new `Double` object that represents the specified `double` value
`double doubleValue()`	Returns the value of this `Double` as a `double`

-a double cannot be directly assigned to an int

-can hold in the range 1.7976931348623157*10^308 to 4.9406564584124654*10^-324

```
Double t = new Double(1.34);
int o = 1.483803;
```

^typo here, the second line should start with “double” not “int”

Char:

-boxed expression : (java.lang.Character)

a. Primarily used for generics, such as defining what type of objects are going in an Arraylist (java.util.ArrayList)

-represents a single character (ASCII or Unicode, but will be only ASCII in APCSA)

a. Thus meaning a char can be easily cast to an Integer to get the ASCII value of that char

-an unsigned 16-bit integer (unsigned meaning it is never negative) and is in the range of: 0 to 65,535

-declared with single quotes (‘)

```
Character c = new Character('x');
char x = 'r';
```

Boolean:

-boxed expression : (java.lang.Boolean)

a. Primarily used for generics, such as defining what type of objects are going in an ArrayList (java.util.ArrayList)

-represents a true or false and only these two values

-cannot be cast to another type without doing some like:

```
boolean t = true;
int x;
if(t) x = 1;
else x = 0;
```

-initialized default with false

```
Boolean y = new Boolean(false);
boolean x = true;
```

String*:

-boxed expression : (java.lang.String)

b. Primarily used for generics, such as defining what type of objects are going in an ArrayList (java.util.ArrayList)

-strings are immutable, meaning they are READ-ONLY

-escape codes such as \n \" \' \\ can be used to print special characters or to override certain things like quotes

-represents an array or list of characters to maybe form a word, phrase, etc.

`String(String str)`	Constructs a new `String` object that represents the same sequence of characters as `str`
`int length()`	Returns the number of characters in a `String` object
`String substring(int from, int to)`	Returns the substring beginning at index `from` and ending at index `to - 1`
`String substring(int from)`	Returns `substring(from, length())`
`int indexOf(String str)`	Returns the index of the first occurrence of `str`; returns `-1` if not found
`boolean equals(String other)`	Returns `true` if `this` is equal to `other`; returns `false` otherwise
`int compareTo(String other)`	Returns a value `<0` if `this` is less than `other`; returns zero if `this` is equal to `other`; returns a value `>0` if `this` is greater than `other`

-checking content between strings should always use the .equals() method

```
String s = new String("hello");
String l = "world";
System.out.println(s.equals(l));
```

String Literal vs String Object:

```
String s = new String("hello"); // object
String l = "world"; // literal
```

-literals with the same content will point to the same literal in memory (shared)

-objects will create new strings no matter if the memory pool already has similar strings (heap)

Complex Types:

-Complex Types (what I call them :/) are things like 2D arrays, 1D arrays of objects or primitives.

-to access a member of the array we do Array[index] where index is from 0 to arr.length - 1

Note: specifying the array length is from 1 to n while specifying index is from 0 to arr.length - 1

-An array is just a list of objects/primitives

-Arrays are declared with []:

```
int[] arr = new int[3];
```

^the number at the end specifies the size of the array, so this array declared can hold 3 values

-going out of bounds would throw an IndexOutBoundsException (AKA ArrayIndexOutOfBoundsException)

-to get the length of the array, we use Array.length

-Array of Objects:

```
Object[] arrObj = new Object[60];
```

^holds 60 objects in total

-go to the loop section to see how you can modify or get values from primitive arrays

-to swap things within an array, we need to create a temporary variables, because the original swap will wipe the old value:

```
int temp = arr[0];
arr[0] = arr[arr.length - 1];
arr[arr.length -  1] = temp;
```

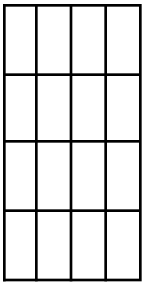
^this swaps the start and the end of the array

-2D arrays are just regular 1D arrays stacked on top of each other (with cols and rows)

-they are declared with two []:

```
int[][] matrix = new int[4][4];
```

-this could be visualized as:



-a 4x4 2D array is this
-to get the rows length, we do Array2D.length
-to get the cols length, we do Array2D[row].length (in the AP-CSA subset, all 2D arrays are rectangular, so you most of the time would be doing Array2D[0].length)
You could go to a 3D array or an nD array, but those are not covered on this AP curriculum, so I am not going over it

Methods:
-methods are used to perform repetitive actions
-they can be static or non-static (below for def)
-methods can be overridden (from inheritance) unless they are declared as final
-methods can have a return type, but if not, they are "void" (you can still type the keyword "return" without a value to jump out of the method)

```
public int sum(int a, int b) { return a + b; }
```

Casting & Operations :

Custom Classes:
Math → abs, pow, random, sqrt
abs → absolute value: Math.abs(...)
Pow → raises a value to a power Math.pow(base, power/exponent)
Random → returns a (pseudo) random between 0.0 and 1.0 exclusive
Sqrt → square root Math.sqrt(...)

Math.random between a start and a finish:

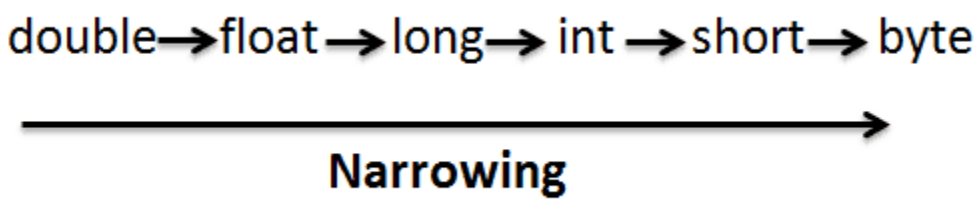
```
int size = end - start + 1;
int rnd = (int) (Math.random() * size) + start;
```

Final:
-variables with final cannot have their value modified
-methods with final cannot be overridden
-classes with final cannot be inherited (extension)

Rounding:
-to round an integer up, we can do:

```
int i = (int) (x + 0.5);
```
-to round down an integer, we can do:
```
int i = (int) (x - 0.5);
```

Explicit & Implicit Casting:
-explicit casting is when the type to cast to is EXPLICITLY STATED:
```
int x = (int) 0.43;
double x = (double) 3;
```
-implicit casting is when the compiler handles the casting automatically and follows this tree of who gets the upper cast:



(int is lower than double, which means whenever a double and an int are in the same expression, the final value would always be a double)
-this means int (+,/,-,*,%) double → double

Operations:
-assignment is done with the "=" symbol
-arithmetic operations are:
+,-,*,/,%
"+" --> Addition, adds two numbers together
"-" → Subtraction, subtracts two numbers together
"*" → Multiplication, multiplies two numbers together
"/" → Division, divides two numbers together
"%" → Modulo, divides two numbers together and returns the remainder
-incrementing and decrementing are represented by:
"--" → Subtract 1 to the variable; decrement per 1
"++" → Add 1 to the variable; increment per 1

```java
int r = 10, int f = 5;
int t = r + f; // 15
int t1 = r - f; // 5
int t2 = r * f; // 50
int t3 = r / f; // 5
int t4 = r % f; // 0
int i = 0;
i++; // i = 1
i--; // i = 0
```

*Refresher on what a remainder, it is the amount left over after division, so for example: 10%4 would give us 2 because 4*2 = 8, 10-8 = 2, which is the amount left over: 2. Another good usage of the modulo operation is to check if a number is even or odd:*

```java
if(x % 2 == 0) System.out.println("EVEN");
else System.out.println("ODD");
```

-compound operations are the standard arithmetic operations by along with the assignment operator:
"+=" add and assign value
"-=" subtract and assign value
"/=" divide and assign value
"*=" multiple and assign value
"%=" mod and assign value
However, compound operators are basically just:

```java
int i = 2, j = 3;
j += i; // j = 5, i = 2
```

And is the same as doing the last line as:

```java
j = j + i; // j = 5, i = 2
```

-you can also concat or append two strings together with "+" or "+=", "+" can be used during the creation of a string to put multiple variables together (no matter type)
-an arithmetic error can occur if one tries to divide by zero, this is known as a ArithmeticException "Divide by zero"

Overloading:
-overloading is when there are multiple methods or constructors with the same name, but different parameters

Logical Operators :

-Comparison between values (primitives)
-returns a boolean if the specified operands are within the logic operations
-the curriculum covers:
<,>,==,!=,>=,<=,!

“>” - greater than; exclusive
“<” - less than; exclusive
“==” - is equal to (NOTE: USE the `equals()` METHOD WHEN COMPARING STRINGS OR OBJECTS)
“!=” - is not equal to
“!” - NOT (logical NOT not bitwise NOT [~])
“>=” - greater than or equal to; inclusive
“<=” - smaller than or equal to; inclusive

```
int i = 0, r = 3;
boolean x = false;
boolean t = i < r; // true, 0 < 3
boolean t2 = i > 5; // false, 0 is not > 5
boolean t3 = 4 != i; // true, 4 is not = 0
boolean t4 = i == 0; // true, 0 = 0
boolean t5 = !x; // true, reversing X (prev false) to true
```

IF/IF-ELSE/IF-ELSE_IF

-if statements follow that the parameterized argument is TRUE (boolean), the code piece in the brackets would be run:

```
if(10 % 2 == 0) System.out.println("Hi");
```

^this would print Hi because, 10 % 2 is equal to 0, thus is true

-if-else statements provide an alternative passageway if the previous if-statement fails (FALSE):

```
if(7 % 2 != 0) System.out.println(":(");
else System.out.println(":)");
```

^this would print :(because 7 % 2 is not 0, however if is it was 10 % 2 instead of 7 %2, it would print :) because it meets the requirements

-if-else_if provides multiples passageways for if the user wants to have different results for specific conditions, an else can also be used in conjunction to operate on else cases:

```
int i = 0;
if(i == 0) System.out.println("ONE");
else if (i == 1) System.out.println("TWO");
else System.out.println("SOMETHING_ELSE");
```

^this would print “ONE”, however if i was equal to 1, it would print “TWO”, otherwise it would print “SOMETHING_ELSE”

Logical Expressions:

-anything that uses a logical operator:

```
4 % 3 == 0
```

a. this would be a raw logical expression

```
if(10 % 2 == 0) {

}
```

a. The logical expression would be 10 % 2 == 0

Combining Logical Expression:

-we can combine multiple logical expressions together using:

&&, ||

"&&" - Logical AND (not bitwise AND [&]), two sides of this operator must be TRUE

```
int i = 3, r = 2;
if(i == 3 && i == 2) {
        System.out.println("HI");
}
```

^here i MUST be 3 AND r MUST be 2 for the program to print HI

"||" - Logical OR (not bitwise OR [|]), any one of the sides that is TRUE

```
int i = 3;
if(i == 3 || i == 4) {
        System.out.println("HI");
}
```

^here i can be EITHER 3 or 4, for the program to print HI

-along with this, we can put logical expressions within logical expressions as inside of parenthesis, doing this deeply would mean we need to follow DeMorgan's Law:

Distribution of Logical/Boolean Operators "DeMorgan's Law"

!(X && B) → !X || !B

!(X || B) → !X && !B

Extras:

X || (C && R) → (X || C) & (X || R)

X && (C || R) → (X && C) || (X && R)

-a good way to do this is thinking of && like multiplication (*) and || as addition (+)

Nested Logical IF:

-you can put IF statements within each other:

```
int i = 3, r = 6;
if(i == 3) {
        if(r == 6) {
                System.out.println(":D");
        } else if (r == 7) {
                System.out.println("D:");
        } else {
                System.out.println("G");
        }
}
```

Loops :

-loops are used to perform actions that occur for either an undefined amount of times or a defined amount of times without having to be repetitive
-there are 3 main types of loops:
Do-While - Similar to a regular While Loop, checks the conditions late
While - Checks the condition before running
For - Has a start, condition, and an increment
For-Each (Enhanced For Loop)

While & Do-While loops run until the parameterized logical expression becomes false

For -Loop structure:
For(start index, condition/end constraint, increment) { }

- Start-index is most often an int declared as the start index (0, or some other number)
- Condition is usually a boolean/logical operator to specify the index constraint
- Increment is a compound arithmetic operator to move the index along instead of being stagnant (usually in the form of index++ or index–)

Example:

```
for(int i = 0; i < 6; i++) {

}
```

^starts from 0 goes to 5 (< is exclusive), goes up by 1

Usages (Looping through an array):

```
int sum = 0;
for(int i = 0; i < arr.length; i++) { // arr is not show, assume its 1D int array
        sum += arr[i];
}
```

^this method of going through an array allows modifying and read values within an array

Sometimes, if we only need to look at the values of an array/arraylist, we can use a for-each loop:

```
int sum = 0;
for(int x : arr) {
        sum += x;
}
```

^this method does not allow us to modify the value, and we do not know where we are in the array

Note to jump out of the loop, we use the "break" keyword.

2D Arrays:

```
// Regular For-Loop
for(int i = 0; i < arr2D.length; i++) {
        for(int j = 0; j < arr2D[0].length; j++) {

        }
}

// For-Each Loop
for(Object[] i : arr2D) {
        for(Object j : i) {

        }
}
```

Recursion:

A method that calls itself many times in order to accomplish something

-must have a valid base case in order to not throw an exception (in particular a StackOverflowException)

a. A base case is the condition in which the recursion would be able to exit

-

Objects & Classes (+Design) :

-objects are instances of classes and has certain values/attributes

-a class defines the implementation of an object; blueprint

Objects (java.lang.Object):

-declared with the new keyword

-all user created classes and instance objects are children of java.lang.Object

-to check the equality between 2 objects, we use Object.equal(Object args) method

-an Object is declared by default as null and if tried to use without any guards to prevent it being null, such as:

```
if(someObj == null) { System.out.println("NULL"); }
```
Will throw a NullPointerException

Global & Local:
-variables can only be used in the most outer brackets that they are found at

Global: declared at the top of the class and can be used at anywhere in the class
Local: can only be used in the current brackets block

Visibility Modifiers:
Private, public

Data Encapsulation → "A pillar of OOP that protects data from being accessed or modified by any part of a program, except with explicit calls to the accessor methods and mutator methods"

Private → non-accessible and only within the class itself; not exposed
Public → Accessible; exposed (a variable without a visibility modifier is automatically marked as public)

Static & Non-Static:
Static → Belongs to the class
Non-Static → Belongs to the object instance

Class Design:
Instance variables: Should be declared as private and non-static variables
Constructors:

- If not declared (At all), Java will default to a no parameter constructor and will not assign anything.
- Constructs the object with attributes
- No Return type
- Has the same name as the class

```
public class ClassName {
        private String str;
        public ClassName(String str) {
                this.str = str;
        }
}
```

!!!! THIS keyword references the current instance, mostly used to reference something from the current class, for example another constructor this(...) or a variable "this.variableName"

Getter methods:

- Methods that give back an instance variable, however this is of course read only, only letting the value be seen, not modified
- Has the word "get" in the method name

Setter Methods:

- Methods that take a value to set to an instance variable
- Is used in conjunction with Getter Methods
- Are Write Only and are void methods.

```
public class ClassName {
        private String str;
        public ClassName(String str) {
                this.str = str;
        }

        public void setStr(String s) {
                str = s;
        }

        public String getStr() {
                return str;
        }
}
```

A good note is to name your getter and setter methods with the same name, except for the get and set part. The name must include only the instanceVariable name and the identifier get or set (in order to not inhibit the meaning).

Common Algorithms :

Linear Search, Binary Search, Selection Sort, Insertion Sort, Merge Sort

HOW TO RECURSION

TRACE THE CODE!!!

Class Mechanics & Inheritance & Polymorphism :

-there can be parent classes and child classes

-we use the "extend" keyword to define that a class will be a child of a parent class:

```
public class ChildClass extends ParentClass {}
```

-the super keyword is called in the constructor ON THE FIRST LINE if needed to specify any parameters needed to be passed to the parent
-super keyword behaves similarly to this keyword

Extension:
-when a child class extends a parent class, all the parents class are automatically inherited

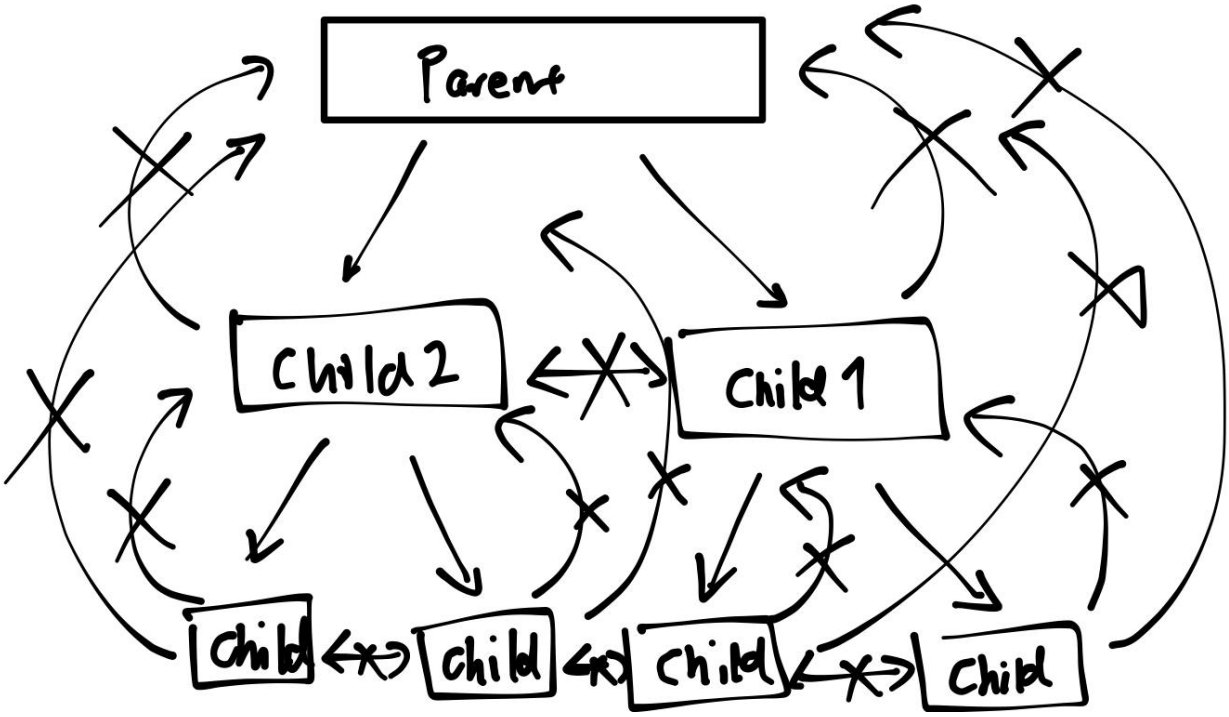
a. This means that within the child class we can call any method from the parent class
b. All methods inherited are overridden if the child class have redefined them, meaning the methods have new content, the child class's implementation would be called unless super is called on the method, then the parents method is called and then the child

```
// in parent class
public void print() { System.out.print("Hi"); }

// in child class
@Override
public void print() {
        super.print();
        System.out.print("World");
}
```

^this would print if the child class's print was called first: "HiWorld"

c. WE CANNOT CALL CHILD CLASS METHODS FROM THE PARENT CLASS



```
Parent e = new Parent(); // allowed
Parent e1 = new Child(); // allowed
Child e2 = new Child(); // allowed
Child e3 = new Parent(); // not allowed
```

-if you used line 2 from ^ above, and you want to call a child method, you need to explicitly cast that object:

```
((Child)e1).childMethod();
```

ArrayList & Primitive Data Structures :

-imported as java.util.ArrayList and/or java.util.List
-List is an interface
-declared as:

```
// you really dont need to specify type on the right side
ArrayList<Object> arrList = new ArrayList<Object>();
```

^you can replace Object with another Object type or another Object
-for primitives, use wrapper classes (Double, Integer, Character, Boolean)
Common methods in an arraylist :

```
ArrayList<Object> arrList = new ArrayList<Object>();
arrList.size(); // gets the length of the array
arrList.add(someObj); // adds an object at the end of the array
arrList.add(index, someObj); // inserts an object at an index
arrList.get(index); // retrieves something at an index
arrList.remove(index); // removes the object at index
arrList.set(index, someObj); // replaces the object at index with someObj
```

NOTE WHENEVER YOU REMOVE SOMETHING FROM THE ARRAYLIST, ALL THE ITEMS IN THE ARRAYLIST SHIFT DOWNWARDS, THUS MAKE SURE TO KEEP YOUR INDEX TRACKING CORRECT

Using loops:

```
// Regular For-Loop
for(int i = 0; i < arrList.size(); i++) {
        System.out.println(arrList.get(i));
}

// For-Each Loop
for(Object j : arrList) {
        System.out.println(j);
}
```