



STUDYRESOURCES

DISCORD

AP EXAM 2020

Notes

<https://t.me/studyresources3>



AP Computer Science A

Exam Date: Friday, May 15, 2020

Question 1 (25min): 65% of the score, Array/ArrayList (Unit 6, 7, 8?), “FRQ1”

Question 2 (15min): 35% of the score, Methods and Control Structures (Unit 2), “FRQ4”

Table of Contents

Click on the topic you want to learn about in the table of contents and click on the link which will take you to the page it is on.

AP Computer Science A	1
Table of Contents	2
Unit 1: Primitive Types	5
1.1: Why Programming? Why Java?	5
S.O.P. vs S.O.P.In:	5
1.2: Variables and Data Types	5
Int:	5
Boolean-:	5
Double:	5
Difference between declaring and initiating variables:	5
1.3: Expressions and Assignment Statements	6
1.4: Compound Assignment Operators	6
1.5: Casting and Ranges of Variables (Implicit and Explicit Type Casting)	6
Unit 2: Using Objects	7
2.1 Objects: Instances of Classes	7
Object:	7
Object reference:	7
Object Class:	7
2.2 Creating and Storing Objects (Instantiation)	7
null reference	7
this reference	7
Instantiation	8
Reference assignment	8
static modifier	8
2.4 Calling a Void Method with Parameters	9
Overloaded methods:	9
Void method with parameters:	9
Scope:	9
Local variable:	9
Implicit parameter:	10
Aliasing:	10
Parameters:	10

2.5 Calling a Non-Void Method	10
2.6 String Objects: Concatenation, Literals, and More	10
Literal Strings	10
Concatenation	11
Immutability	11
Empty Strings	12
Constructors	12
Methods — length, charAt, deleteCharAt	13
Methods — substring	13
Methods — concatenation	13
Methods — Find (indexOf)	13
Methods — Comparisons, Equals, To ignore	14
Methods — Replacements, Uppercase, Lowercase, Trim	14
Methods — Numbers to Strings	14
Methods — Contains	15
Methods — Math, Math Array	15
Methods — Starts or Ends with	15
Methods — Character	17
StringBuffer class	18
Review	18
2.8 Wrapper Classes: Integer and Double	19
Integer class	19
Double class	20
Note	20
Auto-Boxing and -Unboxing	20
2.9 Using the Math Class	20
static int abs(int x)	20
static double abs(double x)	21
static double pow(double base, double exp)	21
static double sqrt(double x)	21
static double random()	21
Math.random	21
Unit 3: Boolean Expressions and if statements	21
3.1 Boolean Expressions	21
3.2 if Statements and Control Flow	22
3.3 if-else Statements	22
3.4 else if Statements	23
3.5 Compound Boolean Expressions	23

3.6 Equivalent Boolean Expressions	23
3.7 Comparing Objects	24
Unit 4: Iteration	24
4.1 While Loops	25
4.2 For Loops	25
4.3 Developing Algorithms Using Strings	26
4.4 Nested Iteration	26
4.5 Informal Code Analysis	26
Unit 5: Writing Classes	26
5.1 Anatomy of a Class	27
5.2 Constructor	27
5.3 Documentation with Comments	28
5.4 Accessor Methods	29
5.5 Mutator Methods	29
5.6 Writing Methods	30
5.7 Static Variables and Methods	32
5.8 Scope and Access	33
5.9 this Keyword	33
5.10 Ethical and Social Implications of Computing Systems	34
Unit 6: One-Dimensional Arrays	34
6.1 Array Creation and Access	34
6.2 Traversing Arrays	35
6.3 Enhanced for Loop for Arrays	36
6.4 Developing Algorithms Using Arrays	36
Unit 7: ArrayList	37
7.1 Introduction to ArrayList	37
7.2 ArrayList Methods	38
The Methods of List<E>	39
7.3 Traversing ArrayLists	40
7.4 Developing Algorithms Using ArrayLists	41
7.5 Searching	41
7.6 Sorting	42
Scanner class:	43
Reference Sheet	44

Unit 1: Primitive Types

1.1: Why Programming? Why Java?

S.O.P. vs S.O.P.ln:

```
System.out.print("Hello World"); //prints out "Hello World" on line 1
System.out.println("Hello World 2"); //still prints out "Hello World 2" on line 1, but now moves
the cursor to line 2
System.out.print("Hello World 3"); //now prints out "Hello World 3" on line 2
```

1.2: Variables and Data Types

Int:

Integers, no decimals
Ex: int price = 1;

Boolean-:

Either "true" or "false"
boolean yes = "true";

Double:

Similar to int, but now with decimals
double price = 1.0;

Difference between declaring and initiating variables:

Declaring(just variable type and name): int price;

Declaring and Initiating(adds a value to the variable type and name): int price = 0;

You must declare and initialize variables before being able to use functions such as "==, <, >, !="

String to int: String one, String two

- int num1Converted = **Integer.parseInt(one)**;
int num2Converted = **Integer.parseInt(two)**;
return num1Converted + num2Converted;
- there is a **method Character.isDigit()** that takes a char as an argument and returns a boolean value
 - If you want to find out if the String is an int or not (ex: "123" is, but "hello" isn't)

1.3: Expressions and Assignment Statements

Remember PEMDAS from 5th grade?

Basic math: $5*2+1$ //result is 11

But now replace the E(exponent) with modulus(M), so PMMDAS (order of operations are: parenthesis first, then either modulus, multiplication, or division (left to right), and finally addition and subtraction” Modulus(%) is basically doing division but instead of getting the usual answer as the answer, you get the remainder as the answer

Division: $25/5$ //result is 5

Modulus: $25\%5$ //result is 0, $34\%3$ //result is 1

Modulus is extremely useful if you what to figure out if an number is even or odd

if number % 2 equals zero, number is even, if number % 2 equals one, number is odd

When assigning values to variables, always remember the value on the right goes into the variable on the left

Int m = 1; int n = 2, m = n //means the value of n (2) now replaces the left int value of m (now 2)

1.4: Compound Assignment Operators

You can use “+=, -=, *=, or /=” to reduce the words written, (useful later when using for and while loops)

$n+=5$ means $n=n+5$, $n*=10$ means $n=n*10$

You can also use “++”, for the same reason(also useful later for and while loops)

$n++$ means $n=n+1$

1.5: Casting and Ranges of Variables (Implicit and Explicit Type Casting)

Making a variable behave as a variable of another type

You can never cast to and from booleans (and later to and from objects like Strings)

No need to cast from an int to a double if a double is already included in the expression, automatically done. This is referred to as implicit type casting.

$\text{double bob} = 1 + 2.0$ //results in a double (3.0)

Not possible to cast from an int to a double if no double exists in the expression (less to more complicated)

$\text{double bob} = 1 + 1$ and $\text{double bob} = (\text{double})(1 + 1)$ //will not run

However, it is possible to cast from double to int (more to less complicated) using “(int) (expression goes in here)”, decimals are removed. This is referred to as explicit type casting.

$\text{Int bob} = (\text{int})(1 + 2.0)$ //after casting would results in a int (3)

Unit 2: Using Objects

2.1 Objects: Instances of Classes

Object:

something that is being created or manipulated by the program

- Characterized by state and behavior
- An object is an idea that corresponds to some real-world object that is being represented by the program

Object reference:

a variable that represents an object

Object Class:

the superclass of which every class automatically extends

2.2 Creating and Storing Objects (Instantiation)

null reference

- An object reference variable that does not currently point to an object
- `name = null;` / `if(name == null)`
- Following a null reference causes a `NullPointerException` to be thrown

this reference

- Allows an object to refer to itself
- Refers to the object thru which the method is being executed (inside a method)
- Call another constructor
- Variable shadowing
- Call object itself (`System.out`)

Example: For example, you can use this to print the object itself. You can say `System.out.println(this)` and it will call the `toString();` function, which will print the object.

Example: Third example, let's say you would like to simply create a method called "deposit" that adds to a bank account's balance. If the bank account has an instance variable called "balance," the method's

statements do not need to be `this.balance += amount;`, and instead, they can be `balance += amount;`. The use of "this" here is unneeded.

Instantiation

- `className objectName = new className();` // instantiating objects in the main class
- We say that "objectName" is a reference to the object
- `String stringName = new String("stringLiteral");` // String class
- `String stringName = "stringLiteral";` // equivalent to above method for String class

Reference assignment

- For object references, assignment copies the memory location
- `bishop2 = bishop1;` (now pointing to where bishop1 was pointing to)
- `equals` is defined for all objects, but unless we redefine it when we write a class, it has the same semantics as the `==` operator (only tells whether it's pointing to the same object, not its value)
- ```
public boolean equals(Fraction f)
{
 if(numerator == f.numerator && denominator == f.denominator)
 return true;
 return false;
}
```

#### static modifier

- Objects do not need to exist in order to be used or exist
- `ClassName.methodName( parameter )`
- Wrapper classes, `System.out/in`, `Math`
- Can only have other static variables or local variables that are passed in ( no `private int a;` )
- ```
public class AClass
{
    public static int sum( int x, int y )
    {
        return x + y;
    }
}
```
- `int k = AClass.sum(3, 4);`

2.3 Calling a Void Method

Void method: don't return a value and have 'void' in the method header instead of the data type

- Mutator methods are often void methods as they change or modify a variable inside the method body instead of returning a value
- Calling a void method (from a different class): `objectName.methodName();`
- Ex:
 - `foo.addToBalance();` // adds 50 to balance each time method is called
- Calling a void method (in the same class): `methodName();`
- Ex:
 - `addToBalance();`

2.4 Calling a Void Method with Parameters

Overloaded methods:

two or more methods in the same class (or a subclass of that class) that have the same name but different parameter lists

- Method's signature: consists of the method's name and a list of the parameter types
- The return type of a
- method is irrelevant

Void method with parameters:

have values passed in their method body, but do not return a value

- Mutator methods can also be void methods with parameters
- Calling a void method (from a different class): `objectName.methodName(parameter1, parameter2, parameter3, ...);`
- Ex:
 - `foo.subtractFromBalance(100);` // subtracts specified amount (100) from
// balance
- Calling a void method (in the same class): `methodName(parameter);`
- Ex:
 - `subtractFromBalance(100);`

Scope:

the region in which the variable or method is visible and can be accessed

Local variable:

defined inside a method

- The scope extends from the point where it is declared to the end of the block in which its declaration occurs

Implicit parameter:

an instance method is always called for a particular object (referred to with the keyword `this`)

Aliasing:

having two references for the same object

- Changing the object information will change the other's reference too
- To copy values, there must be a new object declared
- Ex: `Date birthday = new Date(d.getMonth(), d.getDay(), d.getYear());`

Parameters:

passed by value

- For primitive types, this means that when a method is called, a new memory slot is allocated for each parameter
- The value of each argument is copied into the newly created memory slot corresponding to each parameter
- During the execution of the method, the parameters are local to that method
- Any changes made to the parameters will not affect the values of the arguments in call

2.5 Calling a Non-Void Method

Calling a Non-Void Method: differs from the way a void method is called in that the call is made from within another Java statements

- It always returns a value, so that value must be stored in a variable, printed, or returned to a Java control structure or another method
- Accessor methods are often non-void methods
- Ex: `int y = objectName.methodName();` // given that the method returns 50 each time it is called, int y will have the value of “50”

2.6 String Objects: Concatenation, Literals, and More

Literal Strings

- May include “escape” characters
 - `\\` stands for `\`
 - `\n` stands for a newline
- String `s1 = "C:\\jdk1.4\\docs";`

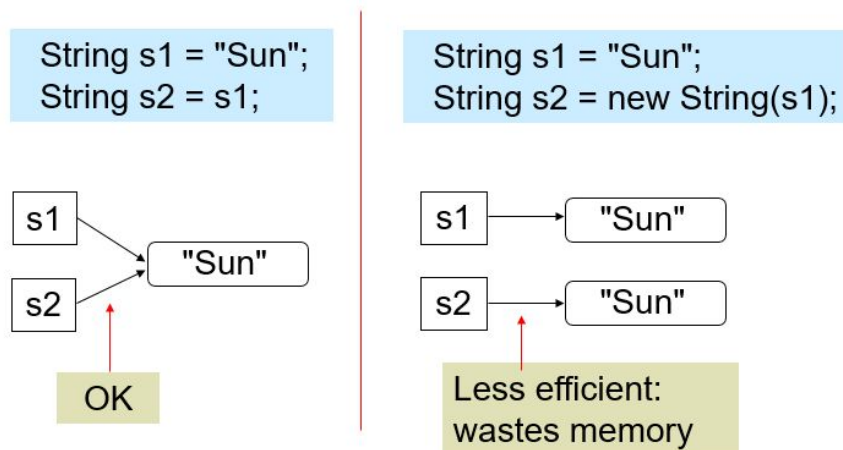
Concatenation

- Concatenation operator: +
- Given two String operands lhs and rhs, lhs + rhs will produce a single String consisting of lhs followed by rhs
 - If either lhs or rhs is an object other than a String, the toString method of the object is invoked, and lhs and rhs are concatenated as before
 - If one of the operands is a String and the other is a primitive type, then the non-String operand is converted to a String and concatenation occurs as before
 - If neither lhs or rhs is a String object, an error occurs
- Example:

```
Date d1 = new Date(8, 2, 1947);           // 8/2/1974
Date d2 = new Date(2, 17, 1948);          // 2/17/1948
String s = "My birthday is " + d2;        // s has value: "My birthday is 2/17/1948"
String s2 = d1 + d2;                      // error: + is not defined for Date objects
String s3 = d1.toString() + d2.toString() // s3 has value: "8/2/19742/17/1948"
```

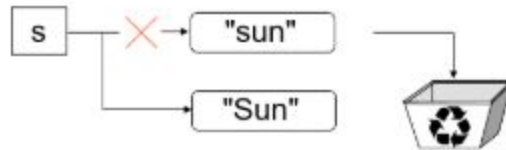
Immutability

- Once created, a string cannot be changed; none of its methods can change the string
 - Immutable objects are convenient because several references can point to the same object safely
 - No danger of changing an object through one reference without the others being aware of the change
- **Advantage: more efficient, no need to copy.**



- **Disadvantage: less efficient** — you need to create a new string and throw away the old one for every small change.

```
String s = "sun";  
char ch = Character.toUpperCase(s.charAt (0));  
s = ch + s.substring (1);
```



Empty Strings

- Has no characters, length = 0;
 - String s1 = "";
 - String s2 = new String();
- Not the same as an uninitialized string
 - private String errorMsg;
 - errorMsg = null;

Constructors

- The constructor's name is always the same as the name of the class. The reason for this is because it specifies the creation of an object of the class. It has no return type.
- No-args and copy constructors are not used much
- String s1 = new String();
- String s2 = new String(s1);
- Example:

```
/* Default Constructor Follows */
```

```
public BankAccount() {  
    password = "";  
    balance = 0.0;  
}  
BankAccount b = new BankAccount();
```

Example 2:

```
/* Constructor with Parameters Follows */

public BankAccount(String acctPassword, double acctBalance) {
    password = acctPassword;
    balance = acctBalance;
}

BankAccount c = new BankAccount("KevinC", 800.00);
```

2.7 String Methods

Methods — length, charAt, deleteCharAt

- Length - str.length() - length of string
- CHAR AT - str.CharAt(str) - Character at given index
- int length(); Returns number of characters in the string
- char charAt(n); Returns the nth char
 - Character positions in strings are numbered starting from 0
- “Flower”.length(); → returns 6
- “Wind”.charAt(2); → returns ‘n’
- “Wind”.deleteCharAt(2); → return “Wnd”

Methods — substring

- String s2 = s.substring(i, j); → returns the substring of chars in positions from i to j-1
- String s2 = s.substring(n); → returns the substring from the nth char to the end
- “strawberry”.substring(2, 5); → returns “raw”
- “emptiness”.substring(9); → returns “” (empty string)

Methods — concatenation

- Concat - str.concat(str) - “cares”.concat(“s”) returns “caress”
- String result = s1 + s2; concatenates s1 and s2
- String result = s1.concat(s2); the same as s1 + s2
- result += s3; concatenates s3 to result
- result += num; converts num to String and concatenates it to result

Methods — Find (indexOf)

- INDEX OF = indexOf(str) - index value of first character in given string
 - If str not found, it will execute -1
- String date = “July 5, 2012 1:28:19 PM”;
- date.indexOf(‘J’); → returns 0
- date.indexOf(‘2’); → returns 8
- date.indexOf(“2012”); → returns 8

- `date.indexOf('2', 9);` → returns 11, starts searching at position 9
- `date.indexOf("2020");` → returns -1, not found
- `date.lastIndexOf('2');` → returns 15

Methods — Comparisons, Equals, To ignore

- **COMPARE TO** - `str.compareTo(str)` - Compare two strings
 - returns 0 if equal,
 - returns a positive value if first mismatch of first string > first mismatch of second string
 - returns a negative value if first mismatch of first string < first mismatch of second string
- **EQUALS** - `s1.equals(s2)` - Use `.equals()` with strings
- `boolean b = s1.equals(s2);` → returns true if the string s1 is equal to s2
- `boolean b = s1.equalsIgnoreCase(s2);` → returns true if the string s1 matches s2, case-blind
- `int diff = s1.compareTo(s2);` → returns the “difference” s1 - s2
- `int diff = s1.compareToIgnoreCase(s2);` → returns the “difference” s1 - s2, case-blind

Methods — Replacements, Uppercase, Lowercase, Trim

- **REPLACE** - `str.replace(old string, new truestring)` - replaces the old string with new string
- **TO LOWERCASE** - `str.toLowerCase()` - Everything will be converted to lowercase characters
- **TO UPPERCASE** - `str.toUpperCase()` - Everything will be converted to uppercase characters
- **Trim** - `str.trim()` - Remove all the white spaces which appear before and after the string
- `String s2 = s1.trim();` → returns a new string formed from s1 by removing white space at both ends
- `String s2 = s1.replace(oldCh, newCh);` → returns a new string formed from s1 by replacing all occurrences of oldCh with newCh
- `String s2 = s1.toUpperCase();`
- `String s2 = s1.toLowerCase();` → returns a new string formed from s1 by converting its characters to upper (lower) case
- Examples:
 - `s1 = s1.toUpperCase();`
 - `s1.toUpperCase();` → s1 remains unchanged
- `String str = “zip”;`
 - `str.replaceAll(“i”, “a”);` → returns “zap”
 - **`str.replaceAll(“z.p”, “zp”) → returns “zp”`**
 - The dot (.) means ‘any char’

Methods — Numbers to Strings

- **Method 1**
 - `String s = “” + n;`
- **Method 2**
 - `String s = Integer.toString(n);`
 - `String s = Double.toString(x);`
- **Method 3:** Can be used to convert str to int in Arrays press [HERE](#)

- String s = String.valueOf(n);
- str[i] = String.valueOf(i); // use in a for loop
- DecimalFormat class can be used for formatting numbers into strings

Example of toString:

```
public String intArrayToString(int[] intArray)
{
    String newWord = ""; //where the loop string [i] will be added

    for(int i = 1; i < intArray.length; i++){ //starts at element 1
        newWord = newWord + ", " + Integer.toString(intArray[i]); //add to list
    }

    if(intArray.length == 0){ //if there are no elements in 'intArray' return []
        return "[]";
    } else { // else return first + newWord
        String first = Integer.toString(intArray[0]); // first element of intArray
        return "[" + first + newWord + "]"; // returns [ + first element + newWord ]
    }
}
```

Methods — Contains

- String date = "July 5, 2012 1:28:19 PM";
- String str1 = date.contains("e"); // false
- String str2 = date.contains("Ju"); // true
- The contains() method checks whether a string contains a sequence of characters.
- Returns true if the characters exist and false if not.

Methods — Math, Math Array

- double a = 12.1; b = 13.1;
- int max = Math.max(a,b); // holds the maximum of two numbers = 13.1
- int min = Math.min(a,b); // holds the minimum of two numbers = 12.1

Methods — Starts or Ends with

- String myStr = "Hello";
- return myStr.startsWith("Hel"); // true
- return myStr.startsWith("o") // false
- return myStr.endsWith("o"); // true
- return myStr.endsWith("llo"); // true
- The startsWith() method checks whether a string starts with the specified character(s).
- Use the endsWith() method to check whether a string ends with the specified character(s).


```
import java.text.DecimalFormat;
...
DecimalFormat money =
    new DecimalFormat("0.00");
...
double amt = 56.7381;
...
String s = money.format (amt);
```

56.7381



"56.74"

- Java 5.0 added printf and format methods:

```
int m = 5, d = 19, y = 2007;
double amt = 123.5;

System.out.printf (
    "Date: %02d/%02d/%d Amount = %7.2f\n", m, d, y, amt);

String s = String.format(
    "Date: %02d/%02d/%d Amount = %7.2f\n", m, d, y, amt);
```

Displays,
sets **s** to:

"Date: 05/19/2007 Amount 123.50"

```
String s1 = "-123", s2 = "123.45";  
int n = Integer.parseInt(s1);  
double x = Double.parseDouble(s2);
```

- These methods throw a **NumberFormatException** if s does not represent a valid number (integer, real number, respectively).

- A safer way:

```
int n;  
do {  
    try  
    {  
        n = Integer.parseInt(s);  
    }  
    catch (NumberFormatException ex)  
    {  
        System.out.println("Invalid input, reenter");  
    }  
} while (...);
```

Methods — Character

- `java.lang.Character`: “wrapper” class that represents characters as objects
- `Character` has several useful static methods that determine the type of a character (letter, digit, etc.)
 - Also has methods that convert a letter to the upper or lower case
- **`if(Character.isDigit(ch))`** ... → return true if ch belongs to the corresponding category
 - `if(Character.isDigit(str.charAt(i)))` FOR IF STAT AND CHARAT
 - `.isLetter...`
 - `.isLetterOrDigit...`
 - `.isUpperCase...`
 - `.isLowerCase...`

- `.isWhitespace...` ← space, tab, newline, etc.
- `char ch2 = Character.toUpperCase(ch1);` / `.toLowerCase(ch1);`
 - if `ch1` is a letter, returns its upper (lower) case; otherwise returns `ch1`
- `int d = Character.digit(ch, radix);`
 - returns the int value of the digit `ch` in the given int `radix`
- `char ch = Character.forDigit(d, radix);`
 - Returns a char that represents int `d` in a given int `radix`

StringBuffer class

- Represents a string of characters as a mutable object
- Constructors:
 - `StringBuffer()` // empty StringBuffer of the default capacity
 - `StringBuffer(n)` // empty StringBuffer of a given capacity
 - `StringBuffer(str)` // converts `str` into a StringBuffer
- Adds `setCharAt`, `insert`, `append`, and `delete` methods
- `toString` method converts this StringBuffer into a String

Review

- What makes the String class unusual?
 - Treated like any type of object yet is not a primitive data type
 - Can be empty
 - Do not have to be created/imported
- How can you include a double quote character into a literal string?
 - `""`
- Is `"length".length()` allowed syntax? If so, what is the returned value?
 - Yes; 6
- Define immutable objects.
 - Objects that cannot be changed or modified
- Does immutability of Strings make Java more efficient or less efficient?
 - Both; immutability makes Java less buggy, but it is also more wasteful because you have to make a new string for every new change
- How do you declare an empty string?
 - `String s1 = "";`
- Why are String constructors not used very often?
 - Strings are immutable, so it's easier to copy a string than copy a reference
- If the value of String `city` is "Boston", what is returned by `city.charAt(2)`? By `city.substring(2, 4)`?
 - s, st
- How come String doesn't have a `setCharAt` method?

- Strings are immutable, so it would return a new string with the request changed character instead of altering the original string
- Is `s1 += s2` the same as `s1 = s1 + s2` for strings?
 - No, because it concatenates `s2` to `s1` instead of forming a new `s1` object
- What do the `indexOf` methods do? Name a few overloaded versions.
 - Returns the position of the first occurrence of the character `c` in the string
 - `s.indexOf('e');`
 - `s.indexOf('e', 4);`
- What is more efficient for strings: `==` and other relational operators or `equals` and `compareTo` methods?
 - `==` only refers to the addresses of the Strings, `equals` and `compareTo` refers to the value of the Strings
- What does the `trim` method do?
 - Deletes whitespace before and after the string
- What does `s.toUpperCase()` do to `s`?
 - Capitalize all letters in String `s`
- What does the `toString` method return for a String object?
 - Returns string representation of the object
- Name a simple way to convert a number into a string.
 - `toString(int n)`, `valueOf(n)`
- **Which class has a method for converting a String into an int?**
 - Java Integer class (`parseInt()`)
- Name a few Character methods that help identify the category to which a given character belongs.
 - `isDigit`, `isLetter`, `isLetterOrDigit`, `isUpperCase`, `isLowerCase`, `isWhiteSpace`
- What is the difference between the String and StringBuffer classes?
 - You cannot change String classes, but you can change the StringBuffer class

2.8 Wrapper Classes: Integer and Double

Integer class

- `Integer (int value)`
 - Constructs an Integer object from an int (boxing)
- `int compareTo(Integer other)`
 - Returns 0 if the value of this Integer is equal to the value of other
 - Returns a negative value if it is less than the value of other
 - Returns a positive value if it is greater than the value of other
- `int intValue()`
 - Returns the value of this Integer as an int (unboxing)

Double class

- Double (double value)
 - Constructs a Double object from a double (boxing)
- double doubleValue()
 - Returns the value of this Double as a double (unboxing)
- int compareTo(Double other)
 - Returns 0 if the value of this Double is equal to the value of other, a negative integer if it is less than the value of other, and a positive integer if it is greater than the value of other
- boolean equals(Object obj)
 - Method overrides equals in class Object
 - Returns true if and only if this Double has the same double value as obj

Note

- Integer and Double objects are immutable; there are no mutator methods in these classes
- If the parameter object for compareTo fails the is-a test, an error will occur

Auto-Boxing and -Unboxing

- There are no primitive types in collections classes; an ArrayList must contain objects, not types like double and int. Numbers must therefore be boxed — placed in wrapper classes like Integer and Double — before insertion into an ArrayList
- Auto-boxing is the automatic wrapping of primitive types in their wrapper classes
- To retrieve the numerical value of an Integer (or Double) stored in an ArrayList, the intValue() (or doubleValue()) method must be invoked (unwrapping)
- Auto-unboxing is the automatic conversion of a wrapper class to its corresponding primitive type. This means that you don't need to explicitly call the intValue() or doubleValue() methods. Be aware that if a program tries to auto-unbox null, the method will throw a NullPointerException.
- Note that while auto-boxing and -unboxing cut down on code clutter, these operations must still be performed behind the scenes, leading to decreased run-time efficiency. It is much more efficient to assign and access primitive types in an array than an ArrayList. You should therefore consider using an array for a program that manipulates sequences of numbers and does not need to use objects.

2.9 Using the Math Class

- static int abs(int x)
 - Returns the absolute value of integer x

- static double abs(double x)
 - Returns the absolute value of real number x
- static double pow(double base, double exp)
 - Returns base exp; assumes base > 0, or base = 0 and exp > 0, or base < 0 and exp is an integer
- static double sqrt(double x)
 - Returns square root of x, $x \geq 0$
- static double random()
 - Returns a random number r, where $0.0 \leq r < 1.0$

Math.random

- You always want to do this: `int num = (int) (Math.random() * 100);` // *100 because it will only give you a number between 0.0 and 1.0;
- Multiplying a number x with Math.random() will result in a random real value y in the range of $0.0 \leq y < x$
- Adding a number x with Math.random() will result in a random real value y in the range of $x \leq y < 1.0 + x$
- In general, to produce a random value in the range $\text{lowValue} \leq x < \text{highValue}$:
`double x = (highValue - lowValue) * Math.random() + lowValue; //for doubles`
`int x = (int)(Math.random() * ((highValue - lowValue) + 1)) + min; //for ints`

Unit 3: Boolean Expressions and if statements

3.1 Boolean Expressions

A Boolean expression is a logical statement that is either TRUE or FALSE . Boolean expressions can compare data of any type as long as both parts of the expression have the same basic data type.

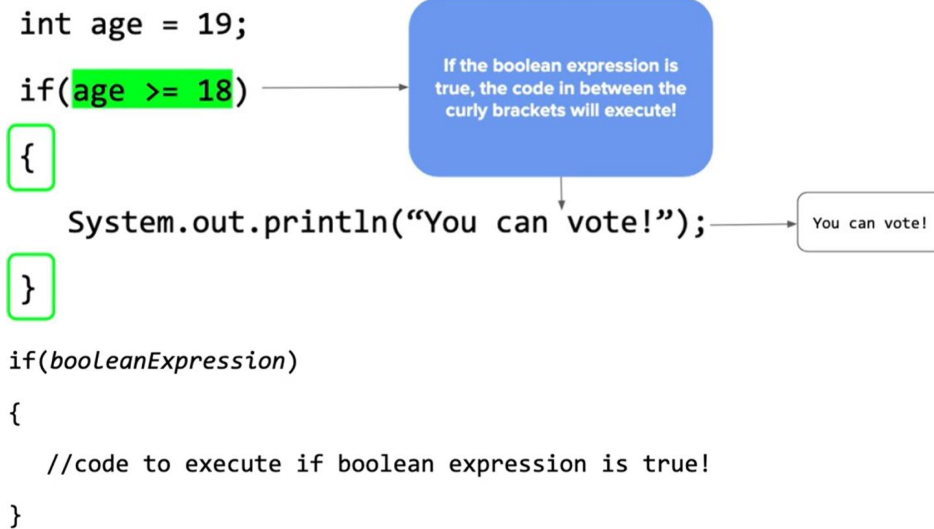
Relational Operator	Sign
Equal	==
Not Equal	!=
Less Than	<
Greater Than	>
Less Than or Equal	<=
Greater Than or Equal	>=

Example:

```
boolean user = true;
if ( user == true) {
    System.out.println("it's true");
}
```

```
} else {  
    System.out.println("it's false");  
}
```

3.2 if Statements and Control Flow



3.3 if-else Statements

```
if(booleanExpression)  
{  
    //code to execute if boolean expression is true!  
}  
else  
{  
    //code to execute if boolean expression is false!  
}
```

3.4 else if Statements

```
if(booleanExpression)
{
    //code to execute if boolean expression is true!
}
else if(booleanExpressionTwo)
{
    //code to execute if else if boolean expression is true!
}
else
{
    //code to execute if all boolean expressions are false!
}
```

3.5 Compound Boolean Expressions

x	y	x y	x	y	x && y
true	true	true	true	true	true
true	false	true	true	false	false
false	true	true	false	true	false
false	false	false	false	false	false

NOT !
 OR ||
 AND &&

3.6 Equivalent Boolean Expressions

- (1.G.1)DeMorgan's Law - In a nutshell, this is a law that will help you solve weird CB boolean MCQs - **The inverse of a boolean statement is equal to that boolean statement.**
 - < becomes >=
 - > becomes <=
 - == becomes !=
 - <= becomes >
 - >= becomes <
 - != becomes ==
- This article explains it well: [5.5. DeMorgan's Laws — AP CSA Java Review](#)
- Now, let's solve an MCQ from the 2014 IPE to practice this.

19. Assume that `a` and `b` have been defined and initialized as `int` values. The expression

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

is equivalent to which of the following?

- (A) `(a != b) || (b < 7)`
- (B) `(a != b) || (b <= 7)`
- (C) `(a == b) || (b <= 7)`
- (D) `(a != b) && (b <= 7)`
- (E) `(a == b) && (b > 7)`

All of the NOTs(!) become true, giving us `(a != b) && (b > 7)`. We then reverse the `!=` to `==`, the `&&` to `||`, and the `>` to `<=`. Therefore, we get `(a == b) || (b <= 7)`, or option C.

`!(A && B) == !A || !B`

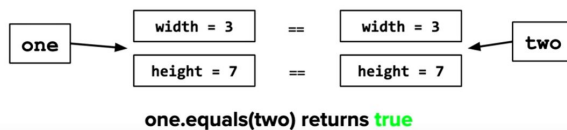
`!(A || B) == !A && !B`

3.7 Comparing Objects

- `.equals()` to compare object values
- We can use `==` and `!=` on objects to determine if the objects are aliases

`.equals()` compares the object values:

```
Rectangle one = new Rectangle(3,7);
Rectangle two = new Rectangle(3,7);
```



```
String str1 = new String("dog");
String str2 = new String("dog");
String str3 = new String("cat");
String str4 = str1;
```

// Print out the results of various equality checks using `equals()`

```
System.out.println(str1.equals(str2)); → true
System.out.println(str1.equals(str3)); → false
System.out.println(str1.equals(str4)); → true
```

Unit 4: Iteration

Before, we actually talk about For loops, and While loops we must understand what an Iteration is; an “Iteration is a technique used to sequence through a block of code repeatedly until a specific condition no longer exists or exists.” So, what that means is that it's pretty much a conditional statement used in java.

There are also “do - while” loops but I don't think that's on the test. (but if you still wanna learn about it: <https://beginnersbook.com/2015/03/do-while-loop-in-java-with-example/>)

4.1 While Loops

While loops are really easy. It's just what they seem to be. What it is that it runs until the condition is false.

Syntax:

```
while(boolean expression)
{
    //will execute if boolean is true, and until
    the boolean expression is false
}
```

Example:

```
Int x = 1;
while( x <= 10) {
    System.out.println("Help!");
    X++;
} // this program will run until x is equal to 10
// So it would run 10 times.
```

4.2 For Loops

Repeating a statement again and again is very redundant, so we use for loops. We use for loops in a variety of ways.

Syntax:

```
for (initialization; test, update) {           // this is the header
    Statement;                                // this is the body
    ...                                        // it's where you write the things repeated
    Statement;
}
```

How does it actually work:

Perform initialization once.

Repeat the following:

Check if the test is true.

If not, stop. Execute the statements.

Perform the update

Example:

```
for (int x = 0; 0 > 10; x++) {  
    System.out.println("Yeet");  
} // this will repeat 10 times.
```

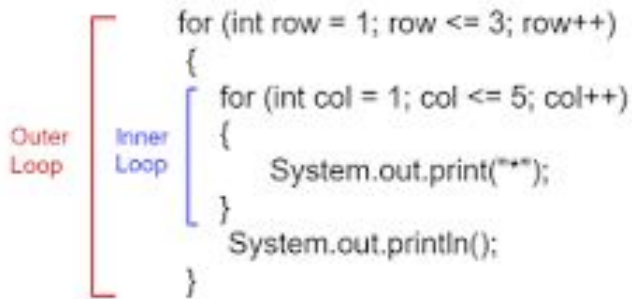
4.3 Developing Algorithms Using Strings

4.4 Nested Iteration

This is pretty much a loop inside another loop. For Example it's a For loop inside a For loop or a While loop inside a While loop or even a While loop inside a For loop. There are literally so many different possibilities.

Textbook Definition: “Nested iteration statements are iteration statements that appear in the body of another iteration statement. When a loop is nested inside another loop, the inner loop must complete all its iterations before the outer loop can continue.”

Example Syntax:



```
for (int row = 1; row <= 3; row++)  
{  
    for (int col = 1; col <= 5; col++)  
    {  
        System.out.print("**");  
    }  
    System.out.println();  
}
```

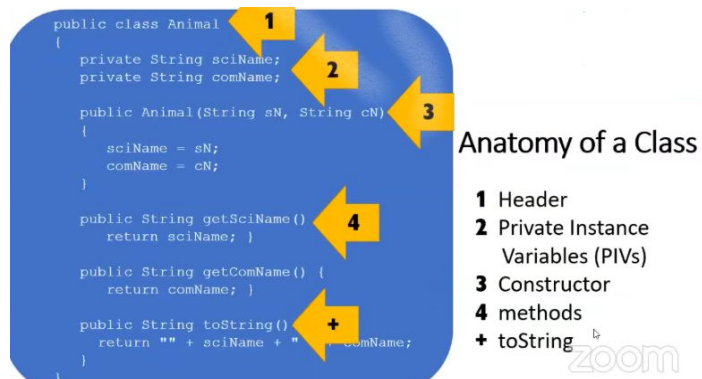
The diagram shows two nested loops. A red bracket on the left side of the first loop is labeled "Outer Loop". A blue bracket on the left side of the second loop is labeled "Inner Loop".

4.5 Informal Code Analysis

Unit 5: Writing Classes

Okay kids, writing classes is going to be essential as on FRQs, there's likely going to be at least one for writing classes, whether it's this or in future years.

5.1 Anatomy of a Class



Public: Allows access from classes outside the declaring class

Private: Restricts access to the declaring class

Classes and constructors are designated as public so that they can be accessed outside of the class file.

What's in a class?

- Instance Variables/Attributes/(Whatever you call it :) Lots of names)
- No Parameter
- Constructor
- Constructors
- Getter
- Setter

Here's a basic example //with comments for your understanding!:

```
public class Frog
{
    private String name;
    private int nEyes;
    private double nLegs;
    public Frog()
    {
        //Constructor
    }

    public static void setName(String n )
    {
        name = n; //Remember, instance variable before the variable given. ALWAYS
    }
}
```

5.2 Constructor

Constructor: creates an object of a class; name is always the same as the class, and has no return type

1. Default constructor: has no arguments and provides reasonable initial values for the object
 2. Constructor with parameters: sets the instance variables of an object to the values of those parameters
 3. Object variables do not store the objects themselves, only the inputs
- Example:

/* Default Constructor Follows */

```
public BankAccount() {  
    password = "";  
    balance = 0.0;  
}  
BankAccount b = new BankAccount();
```

Example 2:

/* Constructor with Parameters Follows */

```
public BankAccount(String acctPassword, double acctBalance) {  
    password = acctPassword;  
    balance = acctBalance;  
}  
BankAccount c = new BankAccount("KevinC", 800.00);
```

5.3 Documentation with Comments

// Single Line Comments

**/* Multi-
Line Comments */**

/ Javadocs
Documentation */**

Javadoc Format:

```
/**  
1. One sentence description of code's function.  
2. Preconditions  
3. Postconditions  
  
4. Block tags  
*/
```

**Multi-Line Comments allow us to easily write additional
comment lines for longer comments:**

```
/* This for loop iterates through a given String,  
and finds all instances of the character 'a' */  
for(int i = 0; i < word.length();i++)  
{  
}
```

**Single Line Comments allow us to add quick comments to
our programs:**

```
//Comments can come before code  
  
int x = 8; //Or directly after  
  
//Or on the line following
```

5.4 Accessor Methods

- Picture duck = new Picture(“ “, 12);

An Accessor method is commonly known as a **get method** or simply a **getter**. A property of the object is returned by the accessor method. **They are declared as public.** A naming scheme is followed by accessors, in other words **they add a word to get in the start of the method name.** They are used **to return the value of a private field.** The same data type is **returned by these methods depending on their private field.**

Syntax

```
01. public int getNumber()
02. {
03.     return Number;
04. }
```

Example

```
01. public class Employee {
02.     private int number;
03.     public int getNumber() {
04.         return number;
05.     }
06.     public void setNumber(int newNumber) {
07.         number = newNumber;
08.     }
09. }
```

- **To call a getter method:**
 - `System.out.println(duck.getName());`

Variable declaration	Getter Method	Setter Method
int quantity	int getQuantity()	void setQuantity(int qty)
String firstName	String getFirstName	void setFirstName(String fname)
Date birthday	Date getBirthday()	void setBirthday(Date Bdate)
boolean rich	boolean isRich() boolean getRich()	void setRich(Booleam rich)

5.5 Mutator Methods

A Mutator method is commonly known as a **set method** or simply a **setter**. A Mutator method mutates things, in other words **changes things.** It shows us the principle of encapsulation. They are also known as modifiers. They are easily spotted because they **started with the word set.** **They are declared as public.** Mutator methods **do not have any return type** and they also **accept a parameter of the same data type depending on their private field.** After that it is used to **set the value of the private field.**

Syntax

```
01. public void set Age(int Age) {  
02.     this.Age = Age;  
03. }
```

public void setAge(int Age){

Example

```
01. public class Cat {  
02.     private int Age;  
03.     public int getAge() {  
04.         return this.Age;  
05.     }  
06.     public void set Age(int Age) {  
07.         this.Age = Age;  
08.     }  
09. }
```

- To call a setter method:
 - duck.setName("Yellow Rubber Ducky");

5.6 Writing Methods

To call a method in Java, write the method's name followed by two parentheses () and a semicolon; In the following example, myMethod() is used to print a text (the action), when it is called:

```
public class Card {
    private String name;
    private String suit;
    private int value;

    //Constructors
    public Card(String name, String suit, int value) {
        this.name = name;
        this.suit = suit;
        this.value = value;

        System.out.println("I am a " + name + " of " + suit + " with a value of " + value + ".");
    }

    //Writing methods
    public void printCard() {
        System.out.println("++++");
        System.out.println("|" + suit + "--|");
        System.out.println("|-" + name + "-|");
        System.out.println("|--" + suit + "|");
        System.out.println("++++");
    }

    //Static method
    public static void printCard(String name, String suit) {
        System.out.println("++++");
        System.out.println("|" + suit + "--|");
        System.out.println("|-" + name + "-|");
        System.out.println("|--" + suit + "|");
        System.out.println("++++");
    }

    //Static method
    public static void myMethod() {
        System.out.println("hello sir India");
    }

    //To String Method
    public String toString() {
        return "Name: " + name + " Suit: " + suit + " Value: " + value + ".";
    }
}
```

```
public class Unit5 {
    public static void main(String[] args)
    {
        //Constructor
        Card card = new Card("K", "S", 10);

        //writing methods
        card.printCard();

        //static methods
        Card.printCard("A", "S");
        Card.myMethod();
    }
}
```

I am a K of S with a value of 10.

```
+++++
|S--|
|-K-|
|--S|
+++++
```

```
+++++
|S--|
|-A-|
|--S|
+++++
```

hello sir India

Calling and creating a method **printArea**


```
public class Rectangle
{
    int width;
    int height;

    public Rectangle(int rectWidth, int rectHeight)
    {
        width = rectWidth;
        height = rectHeight;
    }

    public void printArea()
    {
        int area = width * height;
        System.out.println(area);
    }
}

public class MyProgram
{
    public static void main(String[] args)
    {
        Rectangle rect1 = new Rectangle(3 , 7);
        rect1.printArea();
    }
}
```

A diagram with an arrow pointing from the `rect1.printArea();` line in the `MyProgram` class to the `printArea()` method definition in the `Rectangle` class.

Overloaded methods: two or more methods in the same class (or a subclass of that class) that have the same name but different parameter lists

- Method's signature: consists of the method's name and a list of the parameter types
- Return type of a method is irrelevant

5.7 Static Variables and Methods

```
class JavaExample3 static int var1;
static String var2;
//This is a Static Method
static void disp(){
    System.out.println("Var1 is: "+var1);
    System.out.println("Var2 is: "+var2);
}

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

Var1 is: 0
Var2 is: null

Static variable (class variable): contains a value that is shared by all instances of the class

- Static: memory allocation happens only once
- Used to keep track of statistics for objects of the class
- Accumulate a total
- Provide a new identity number for each new object of the class

Static final variables (constants): cannot be changed and are often declared public

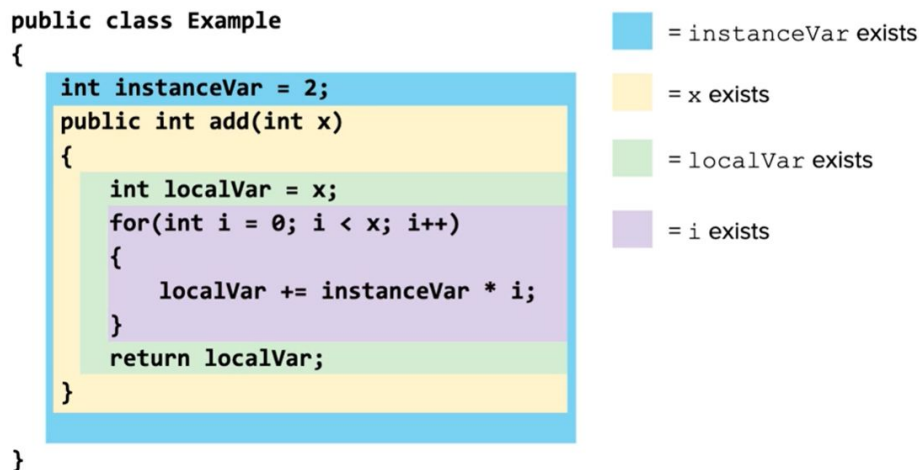
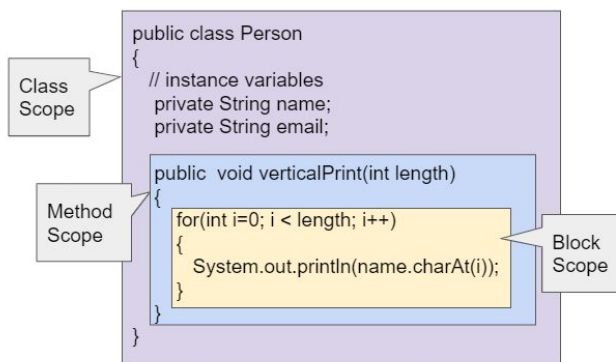
Static method (class method): a method that performs an operation for the entire class, not its individual objects

- Static method cannot call on an instance method or variable
- Invoked by using the class name with the dot operator
- Usually a class creates no objects of the class

5.8 Scope and Access

Local variables only exist in the context of the method or constructor they are created in

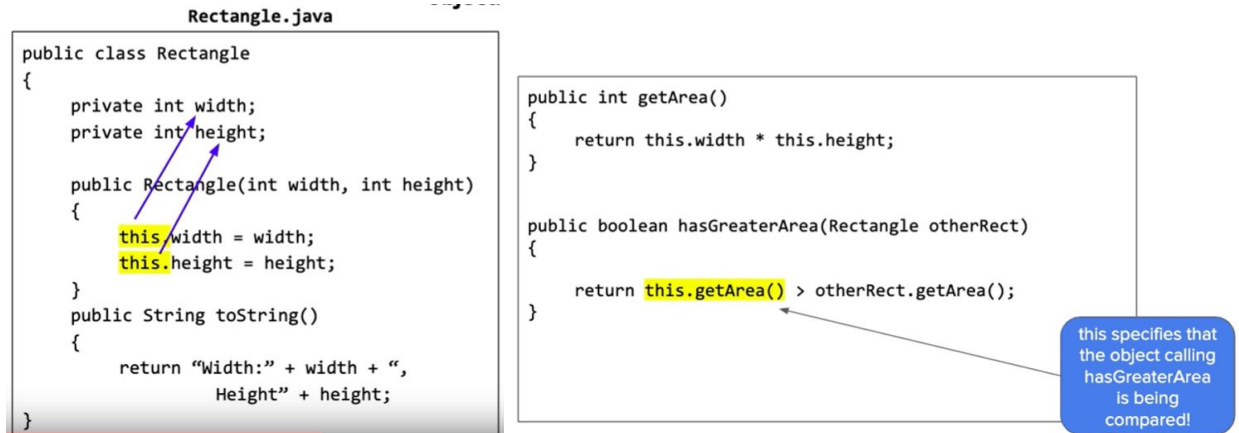
Scope: in general, a variable exists from the point where it is declared, until the end of the block it is declared inside of.



5.9 this Keyword

this is a reference to the current object whose methods and constructors are being called.

this can also be used to call an object's methods.



5.10 Ethical and Social Implications of Computing Systems

While programs are typically designed to achieve a specific purpose, they may have unintended consequences.

System reliability is limited. Programmers should make an effort to maximize system reliability.

Legal issues and intellectual property concerns arise when creating programs.

The creation of programs has impacts on society, economies, and culture. These impacts can be beneficial and/or harmful.

Unit 6: One-Dimensional Arrays

[Good Array Help](#) -- Powerpoint (This might help for those who have no idea on what it is) (It also has links to practice problems)

[Practice Problems](#) -- Let's use this document to add FRQ's related to arrays. (Try to use the same format so it is neat and easy to follow)

6.1 Array Creation and Access

One-Dimensional Arrays

- Array: a data structure used to implement a list object, where the elements in the list are of the same type
- For an array of N elements in Java, index values go from 0 to N - 1
 - If a negative N is used, or a value k where $k \geq N$, an `ArrayIndexOutOfBoundsException` is thrown
- Can be aliased. When setting an array equal to another array without looping through, both arrays point to the same address in memory.

Initialization

- All of the below methods creates an array of 25 double values and assigns the reference data to this array:
 1. `double [] data = new double [25];`
 2. `double data [] = new double [25];`
 3. `double [] data;`
`data = new double [25];`
- Initializer List
 - `int [] coins = {1, 2, 4, 5, 6};`

Length of Array

- `length`: a final public instance variable that can be accessed when you need the number of elements in an array
- Note:
 1. The array subscripts go from 0 to `name.length - 1`; therefore, the test on `i` in the for loop must be strictly less than `names.length`
 2. `length` is not a method and therefore is not followed by parentheses. Contrast this with String objects, where `length` is a method and must be followed by parentheses.For example,

```
String s = "Confusing syntax!";  
int size = s.length();           // assigns 17 to size
```

6.2 Traversing Arrays

Traversing an Array

- Use a for-each loop whenever you need access to every element in an array without replacing or removing any elements
- Use a for loop in all other cases: to access the index of any element, to replace or remove elements, or to access just some of the elements
- Example

1:

```
int count = 0;  
for( int num : arr )  
    if( num % 2 == 0 )  
        count++;  
return count;
```

2:

```
for( int i = 0; i < arr.length; i+= 2 )  
    Arr[i] = 0;
```

Indexes for "for loops" can be: `int i = 0; i < arr.length` or `int i = 0; i <= arr.length - 1`

6.3 Enhanced for Loop for Arrays

For Each Loop Uses:

- Traversing an array or ArrayList
- Read only
- When you don't need an index
- When you only need to access one element at a time
- Traverse: visiting each element sequentially
- for(Declared variable : what you are traversing)

```
int[ ] arr = {2, 4, 6, 8};  
for( int x : arr )  
{  
    System.out.println(x);  
}
```

int x represents the actual element, not its index

Changes to int x does not change the value in the array

Don't use i for for:each because it is misleading

Will go through loop number of arr.size() / arr.length

6.4 Developing Algorithms Using Arrays

Arrays as Parameters

- Since arrays are treated as objects, passing an array as a parameter means passing its object reference; no copy is made of the array
- Thus, the elements of the actual array can be accessed and modified
- Primitive types, including single array elements of type int or double, are passed by value; a copy is made of the actual parameter, and the copy is erased on exiting the method
- Example

1:

```
int min = arr[0];  
int minIndex = 0;  
for( int i = 1; i < arr.length; i++ )  
    if( arr[i] < min )  
    {  
        min = arr[i];  
        minIndex = i;  
    }  
return minIndex;
```

2:

```
for( int i = 0; i < b.length; i++ )  
    b[i] += 3;
```

Array of Class Objects

- The statement

```
allDecks = new deck[NUMDECKS];
```

creates an array, allDecks, of 500 Deck objects. The default initialization for these Deck objects is null. In order to initialize them with actual decks, the Deck constructor must be called for each array element. This is achieved with the for loop of the ManyDecks constructor.
- In the shuffleAll method, it's OK to use a for-each loop to modify each deck in the array with the mutator method shuffle

Unit 7: ArrayList

7.1 Introduction to ArrayList

Array Lists

- An ArrayList provides an alternative way of storing a list of objects and has the following advantages over an array:
 - An ArrayList shrinks and grows as needed in a program, whereas an array has a fixed length that is set when the array is created
 - In an ArrayList list, the last slot is always list.size() - 1, whereas in a partially filled array, you, the programmer, must keep track of the last slot currently in use
 - For an ArrayList, you can do insertion or deletion with just a single statement. Any shifting of elements is handled automatically. In an array, however, insertion or deletion requires you to write the code that shifts the elements.
 - It is easier to print the elements of an ArrayList than those of an array. For an ArrayList list and an array arr, the statement

```
System.out.print( list );
```

will output the elements of the list, nicely formatted in square brackets, with the elements separated by commas. Whereas to print the elements of arr, an explicit piece of code that accesses and prints each element is needed. The statement

```
System.out.print( arr );
```

will produce a weird output that includes an @ symbol and the hashCode of the array in hexadecimal.

- Standard format to create an arrayList: **ArrayList<E> array = new ArrayList<E>();**

- **<E>** is the datatype - can be Integer, String, Double
- We don't need to specify how many items can be stored in an ArrayList -> 0;
- We can actually write: `ArrayList list = new ArrayList();`

Feature	array	ArrayList
Create	<code>int[] arr = new int[5]</code>	<code>ArrayList<Integer> list = new ArrayList<Integer>();</code>
Mutability	Fixed size	Changing size
Types	Can store Primitives or Objects	Can only store Objects, handles autoboxing and unboxing
When to Use	With data sets that don't change in value or size frequently	With data that changes in value or size frequently

7.2 ArrayList Methods

ArrayList Method	Function	Example
<code>boolean add(E obj)</code>	Appends obj to end of list	<code>arrayList.add("Add!");</code>
<code>void add(int index, E obj)</code>	Inserts obj at position index, moves elements index and higher up, and adds 1 to size	<code>arrayList.add(2, "Add");</code>
<code>E get(int index)</code>	Returns element at the position index	<code>String elem = arrayList.get(2);</code>
<code>int size()</code>	Returns the number of elements in the list	<code>int num = arrayList.size();</code>
<code>E set(int index, E obj)</code>	Replaces element at index with obj. Returns replaced element	<code>arrayList.set(2, "NewVal");</code>
<code>E remove(int index)</code>	Removes element from position index. Subtracts 1 from size and moves elements index and higher down one index.	<code>arrayList.remove(2);</code>

The List<E> Interface

- A class that implements the List<E> interface — ArrayList<E> for example — is a list of elements of type E. In a list, duplicate elements are allowed. The elements of the list are indexed, with 0 being the index of the first element.
- A list allows you to
 - Access an element at any position in a list using its integer index
 - Insert an element anywhere in the list
 - Iterate over all elements using ListIterator or Iterator
- When you use `.remove()`, the ArrayList automatically shifts the things in the ArrayList. This is known as the "Delete Skip Bug." One way to avoid this is by starting from the back of the list instead like so:

```
for (int k = thing.size() - 1; k > -1; k--) {
```

You can also do `k--`; after an object is removed from the list which will also fix the Delete Skip Bug

The Methods of List<E>

- `boolean add(E obj)`
Appends obj to the end of the list. Always returns true. If the specified element is not of type E, throws `ClassCastException`
- `int size ()`
Returns the number of elements in the list
- `E get (int index)`
Returns the element at the specified index in the list
- `E set (int index, E element)`
Replaces item at specified index in the list with specified element. Returns the element that was previously at index. If the specified element is not of type E, throws an exception (`ClassCastException`)
- `void add (int index, E element)`
Inserts elements at specified index. Elements from position index and higher have 1 added to their indices. Size of list is incremented by 1
- `E remove (int index)`
Removes and returns the element at the specified index. Elements to the right of position index have 1 subtracted from their indices. Size of list is decreased by 1
- `Iterator<E> iterator ()`
Returns an iterator over the elements in the list, in proper sequence, starting at the first element

The ArrayList<E> Class

- The main difference between an array and an ArrayList is that an ArrayList is resizable during run time, whereas an array has a fixed size at construction
- Shifting of elements, if any, caused by insertion or deletion, is handled automatically by ArrayList; operations to insert or delete at the end of the list are very efficient. Be aware, however, that at some point there will be a resizing, but on average, over time, an insertion at the end of a list is a single, quick, operation; in general, insertion and deletion in the middle of an ArrayList requires elements to be shifted to accommodate a new element (add) or to close a “hole” (remove)

The Methods of ArrayList<E>

- `ArrayList()`
Constructs an empty list
- Note

Each method above that has an index parameter (add, get, remove, and set) throws an `IndexOutOfBoundsException` if the index is out of range. For get, remove, and set, index is out of range if

`index < 0 || index >= size()`

For add, however, it is okay to add an element at the end of the list. Therefore index is out of range if

`index < 0 || index > size()`

- Arrays vs. ArrayLists

Feature	Arrays	ArrayLists
Initialize	<code>int[] arr = new int[5];</code>	<code>ArrayList<Integer> array = new ArrayList<Integer>();</code>
Get Value	<code>arr[index]</code>	<code>array.get(index)</code>
Get Size	<code>arr.length</code>	<code>array.size()</code>
Set Value	<code>arr[index] = value;</code>	<code>array.set(index, value)</code>

7.3 Traversing ArrayLists

Collections and Iterators

- Iterator: an object whose sole purpose is to traverse a collection, one element at a time; during iteration, the iterator object maintain a current position in the collection, and is the controlling object in manipulating the elements of the collection

The `Iterator<E>` Interface

- Package `java.util` provides a generic interface, `Iterator<E>`, whose methods are `hasNext`, `next`, and `remove`

The Methods of `Iterator<E>`

- `boolean hasNext()`
Returns true if there's at least one more element to be examined, false, otherwise
- `E next()`
Returns the next element in the iteration; if no elements remain, the method throws a `NoSuchElementException`
- `void remove()`
Deletes from the collection the last element that was returned by `next`; this method can be called only once per call to `next`; it throws an `IllegalStateException` if the `next` method has not yet been called, or if the `remove` method has already been called after the last call to `next`

Using a Generic Iterator

- To iterate over a parameterized collection, you must use a parameterized iterator whose parameter is the same type
- Example

1:

```
Iterator<String> itr = list.iterator();
while (itr.hasNext())
    System.out.println( itr.next() );
```

Equivalent to →

```
for ( String str : list )
    System.out.println( str );
```

Easiest way to remove all occurrences from an ArrayList is to use an iterator.

2:

```
Iterator<String> itr = strList.iterator();
while (itr.hasNext())
    itr.remove();
```

7.4 Developing Algorithms Using ArrayLists

7.5 Searching

<https://www.youtube.com/user/AlgoRythmics> - Dancing Algorithms! Good visuals

Binary

Pseudocode: Not quite efficient, but for the learning purposes. Only works on SORTED arrays. :) First of all, we have two “searchers” starting at the ends of the array (max and min, hence the name “binary”). We find the middle of the array and compare our search value to it. Is the search value greater? Okay, move the min searcher to mid+1. Is the search value less than? Okay, move the max value to mid-1. Then we repeat with the new min/max. Find a mid between them and compare search value. You should know what happens after.. Ask the two questions...do it... alllll the way until we find that the mid matches the search value. Bingo!

Linear

Pseudocode: Basically, go through each element one by one and compare it to the value that you are looking for. If the current element and the search value are not equal, move on. If they are, the search is done and return the index of where you found the match.

Computers store vast amounts of data. One of the strengths of computers is their ability to find things quickly. This ability is called searching. For the AP CS A exam you will need to know both **sequential** search and **binary** search.

- **Sequential search** typically starts at the first element in an array or list and looks through all the items one by one until it either finds the desired value and then it returns the index it found the value at or if it searches the entire array or list without finding the value it returns -1.
- **Binary search** can only be used on data that has been sorted or stored in order. It checks the middle of the data to see if that middle value is less than, equal, or greater than the desired value and then based on the results of that it narrows the search. It cuts the search space in half each time.

7.6 Sorting

Sorting

- **Selection Sort** - Selected the smallest item from the current location on to the end of the array and swap it with the current position. Do this from index 0 to the array length - 2. You don't have to process the last element in the array, it will already be sorted when you compare the prior element to the last element.
- **Insertion Sort** - Insert the next unsorted element in the already sorted part of the array by moving larger values to the right. Start at index 1 and loop through the entire array.
- **Merge sort** - Break the elements into two parts and recursively sort each part. An array of one item is sorted (base case). Then merge the two sorted arrays into one.

Scanner class:

- Import the scanner class:
 - `import java.util.Scanner;`
- Create a Scanner:
 - `Scanner input = new Scanner(System.in); // Create a Scanner object`
`System.out.println("Enter username");`

`String userName = input.nextLine(); // Read user input`

`System.out.println("Username is: " + userName); // Output user input`
-
- Input types:

Method	Description
<code>nextBoolean()</code>	Reads a <code>boolean</code> value from the user
<code>nextByte()</code>	Reads a <code>byte</code> value from the user
<code>nextDouble()</code>	Reads a <code>double</code> value from the user
<code>nextFloat()</code>	Reads a <code>float</code> value from the user
<code>nextInt()</code>	Reads a <code>int</code> value from the user
<code>nextLine()</code>	Reads a <code>String</code> value from the user
<code>nextLong()</code>	Reads a <code>long</code> value from the user
<code>nextShort()</code>	Reads a <code>short</code> value from the user

Reference Sheet

	<u>ArrayLists</u>	<u>Arrays</u>	<u>Strings</u>
Import	Import java.util.ArrayList;	-----	-----
Declare	ArrayList<String> Al = new ArrayList<String>();	String[] Ar = new String[7];	String Str = "Comp Sci"
Tell How Long	Al.size();	Ar.length;	Str.length();
Get an Element by Index	Al.get(3) //leaves Al.remove(3) //removes	Ar[3];	Str.substring(3,4); Str.charAt(3);
Set an Element by Index	Al.set(3, "Python"); //replaces	Ar[3] = "Python";	-----
Find an Element	Al.indexOf("Python");	-----	Str.indexOf("m");
Add an Element	Al.add("Lisp"); Al.add(2, "Lisp");	-----	Str = Str + "Rules";
Remove an Element	Al.remove("Basic"); Al.remove(0);	-----	-----

Java Quick Reference

Accessible methods from the Java library that may be included in the exam

Class Constructors and Methods	Explanation
String Class	
<code>String(String str)</code>	Constructs a new String object that represents the same sequence of characters as str
<code>int length()</code>	Returns the number of characters in a String object
<code>String substring(int from, int to)</code>	Returns the substring beginning at index from and ending at index to - 1
<code>String substring(int from)</code>	Returns <code>substring(from, length())</code>
<code>int indexOf(String str)</code>	Returns the index of the first occurrence of str ; returns -1 if not found
<code>boolean equals(String other)</code>	Returns true if this is equal to other ; returns false otherwise
<code>int compareTo(String other)</code>	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
Integer Class	
<code>Integer(int value)</code>	Constructs a new Integer object that represents the specified int value
<code>Integer.MIN_VALUE</code>	The minimum value represented by an int or Integer
<code>Integer.MAX_VALUE</code>	The maximum value represented by an int or Integer
<code>int intValue()</code>	Returns the value of this Integer as an int
Double Class	
<code>Double(double value)</code>	Constructs a new Double object that represents the specified double value
<code>double doubleValue()</code>	Returns the value of this Double as a double
Math Class	
<code>static int abs(int x)</code>	Returns the absolute value of an int value
<code>static double abs(double x)</code>	Returns the absolute value of a double value
<code>static double pow(double base, double exponent)</code>	Returns the value of the first parameter raised to the power of the second parameter
<code>static double sqrt(double x)</code>	Returns the positive square root of a double value
<code>static double random()</code>	Returns a double value greater than or equal to 0.0 and less than 1.0
ArrayList Class	
<code>int size()</code>	Returns the number of elements in the list
<code>boolean add(E obj)</code>	Appends obj to end of list; returns true
<code>void add(int index, E obj)</code>	Inserts obj at position index ($0 \leq \text{index} \leq \text{size}$), moving elements at position index and higher to the right (adds 1 to their indices) and adds 1 to size
<code>E get(int index)</code>	Returns the element at position index in the list
<code>E set(int index, E obj)</code>	Replaces the element at position index with obj ; returns the element formerly at position index
<code>E remove(int index)</code>	Removes element from position index , moving elements at position index + 1 and higher to the left (subtracts 1 from their indices) and subtracts 1 from size; returns the element formerly at position index
Object Class	
<code>boolean equals(Object other)</code>	
<code>String toString()</code>	