AP Computer Science A
Ardent Academy Winter Session 2018-2019

Today's Agenda:
1. Student-led Questions/Answers
2. Review of Object-Oriented Programming Fundamentals
3. Topics to date include:
    Classes, Objects, Class Design
    Inheritance, Polymorphism
    Cohesion, Coupling

Three Units
1-Structured Programming
2-Procedural Programming
3-OOP

Many of the concepts you learn are not unique to each unit, but rather exist in all three.

input/output, variables, operators, typecasting, boolean expressions, decisions/conditional control structures, loops, methods, data structures (array and arraylist).

Q/A
1. Method Recursion
2. Global variables, Class Variables, Instance Variables
3. Arrays and ArrayLists -- how to use? when to use?
4. Data Structures -- what about the other ones besides arrays?
5. Code efficiency -- space, organization, code convention
6. Program Design -- procedural abstraction - howto?
7. The toString() method.

--

Recursion

HelloHelloHelloHelloHelloHello
Hello? Anyone there? Helloooo!? Hello. Hellllooo!!!!!
Hello Hello Hello Help! Help! Hello Hello Hello

Recursion is where a function calls itself.

A recursive method contains a call to itself.
This allows it to repeat an action without using a loop.

A recursive function has two parts:
1. base case - this is a conditional control structure (an if statement) that controls whether the function will be allowed to continue.
2. recursive call - this is here the method calls itself

In practice, we use recursive calls to take on a very large problem that can be solved by breaking it up into many small identical pieces. The base case then represents the smallest and most trivial solution. By *stepping in* to the recursion, the multiple calls continue to solve an ever decreasing problem size until we get to 1 or 0. Then, by *stepping out*, we combine the results back together into a final answer.

Recursion is not particularly memory effective, nor is it fast. However, it is much simpler in most cases to represent the recursive version of a solution. Let's look at a few recursive examples.

Triangle Numbers

```
1         = 1
12        = 3
123       = 6
1234      = 10
12345     = 15
123456    = 21
etc...
```

Fibonacci Sequence

The fibonacci sequence is a popular topic in mathematics and series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc...

We understand that each number n of the series is the sum of the previous two numbers:

n = (n-1) + (n-2)

Express this as a recursive function.

As you can see while the recursive function is easy to write, it is not very efficient for very large values of n. We can improve this by writing an iterative version of the same function.

The iterative version mimics how we might write out the answers to this on a sheet of paper. Starting with the first few values of n, it is very trivial to solve because we always know the previous two values. However, we do not need to know anything other than the previous two values. Therefore, when we iterate, it is not necessary to store any values other than the previous two, followed by the next one that we calculate.

Now, the iterative version discards all but the previous two numbers and it does not need to recalculate any previously calculated numbers, allowing it to run at nearly the same speed as any loop.

We do notice that the integer data type is unable to hold the full value of the calculated numbers because int is limited in size to 32 bits. Java gets around this problem with the BigInteger class, which in the documentation, let's you know that even if there were an upper bound for the largest integer that it could hold, none of you owns a computer with enough memory to test this limitation.

If you see a recursion question on the AP test, if it is free response, you will likely have to write code for two steps:

1. identify the base case
2. write the recursive call

If you have trouble trying identify the base case, sometimes a recursive solution requires you to modify the original problem slightly so that it can be simplified through simple subtraction.

For multiple choice, it is common to ask if you can determine the values of the variables after several recursive calls, or if you can determine the output after several rounds of recursion.

--

Global Variables

The concept of a global variable is one that can be seen everywhere in a program, and can be read or modified by anyone. As a matter of convenience, programmers like having global variables because it makes accessing values very simple. As a matter of security or stability, this is a terrible idea because anything in a program can change the variables whether or not that that variable is even relevant to it.

In Java, we can control whether a variable or method can be seen by something inside of a class or outside of a class. By using the **public** keyword, we allow a variable or method to be accessed from outside of the class. By using the **private** keyword, we limit access to that variable or method to other parts inside that class.

There is also a distinction between variables and methods that are accessible directly from a class definition, and those that only exist when an object has been instantiated from that class. We call the first group *class variables* and *class methods*, and the second group *instance variables* and *instance methods.* Class variables and methods have the **static** keyword associated with them.

Arrays and ArrayLists

Java's fundamental data structure for holding more than one element of each type is called the **array.** There are three ways to make an array:

```
// first way - declare, then instantiate
int[] nums;
nums = new int[ 10 ];

// second way - declare and instantiate in one step
int[] nums = new int[ 10 ];

// third way - declare, instantiate, and populate
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
```

Arrays can hold only one kind of data at a time; however they can store any primitive data type or abstract data type:

```
String[] names = { "Bob", "Sally", "Le-a" };
```

Arrays have a major limitation, in that once they have been instantiated to a certain size, that is the size they will be. They cannot increase or decrease in size. In practice, this is a good thing because it makes it clear how much memory an array will use. In some cases, this is a bad thing because it cannot adapt to different amounts of data.

To get around this problem, Java has a built-in data structure called **ArrayList**.

ArrayList is a List that has much of the behaviors of a standard array. However, it is a *dynamic implementation of an array* meaning that it is able to grow or shrink to acommodate the data that is put into it.

```
import java.util.ArrayList;
ArrayList<String> names = new ArrayList<String>();
```

To work with the data inside of an ArrayList, you must use the methods that the ArrayList class provides. The Java online language reference has a

detailed page that covers all of the methods that are available that govern the behavior of this object.

There is one drawback to ArrayList...it cannot by its default self handle primitive data types, as it can only accept abstract data types! To cope with this problem, Java incorporates special *wrapper classes* so that ArrayLists can deal with integers and doubles, and goes through this process automatically in *autoboxing* and *autounboxing* such that the behavior of ArrayList with primitives like integers and doubles is identical to that of a standard array. The classes used are the class **Integer** and the class **Double**.

Data Structures

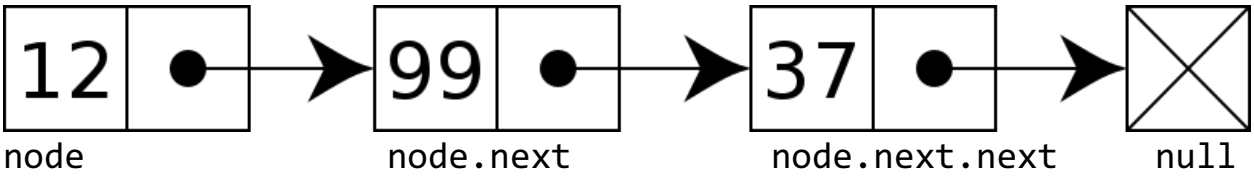
There are several kinds of data structures that we can create that can handle data in unique ways. Some of these have existed in computer science for a long time and we simply learn how to implement them in any given programming language.

One popular data structure is called a **Stack**. A stack is a LIFO (last-in-first-out) data structure, where the most recent item that has been put into it will be retrieved first.

The corresponding data structure is called a **Queue**. A queue is a FIFO (first-in-first-out) data structure which mimics the lines that are used for everything from shopping checkout to theme park rides.

One more interesting data structure is the **Linked List**. Rather than hold anything in a specific kind of container, a linked list contains *nodes* that store the data, along with a *pointer* to the next node.

```
Class Node
{
    Data data;
    Node next;
}
```



This is an example of a linked list of integers. As you can see, each node points to the next one. The first node is called the *head*, and the last node is called the *tail*. The tail points to *null.*

Code Conventions, Readability, and Efficiency

In general, the goal behind the code conventions is to ensure that Java code is uniformly formatted so that it is easy to read, and eliminates unnecessary whitespace or bizarre structures.

In some cases, the curly braces are not necessary, although it doesn't hurt to put them where they are most commonly used. If a decision structure only has a single statement, then the curly braces can be skipped. However, it might be useful to put the statement code on a separate, indented line vs. inline with the condition.

None of this is enforced by the language itself, but has evolved organically over the decades by people who use Java in mission-critical applications and have thus defined a common structure that we all use. Keeping to these code conventions improves the chances that your code will be free from errors, especially as it is easier to debug.

Class Design, Program Design

There aren't any hard and fast rules about class design and program design. Each software project that you start is going to have different requirements and criteria. However, you do want to apply some of the really basic rules in software development:

1. **Cohesion** - cohesion describes how well a class describes a single concept or idea. Software that does not conflate too many concepts or ideas into a single container will perform better with less faults. This is akin to the old expression "don't put all your eggs in one basket." In practice, if the class you are designing seems overly-complicated and difficult to implement, then you probably need to apply procedural abstraction to it AFTER you separate its purpose into multiple classes.

2. **Coupling** - coupling describes the level of or number of *interdependencies* in a software system. Systems with high coupling are unstable because a single change in one part will cause the entire rest of it to malfunction. Systems with low coupling have isolated key elements so that chages in one area will not disrupt how the rest of the system functions.

We aim to write software that is **high cohesion, low coupling**.

We run into terrible software that is **low cohesion, high coupling**. Avoid this whenever possible!

toString() is a method present in the Object class, the "mother of all superclasses" which is inherited by all classes in Java. It returns a string representation of the class it is in. In common use, we *override* the coding of this method to generate a specific string representation of the class we have it written in. So for example, a Circle object would have a string that reflects what the instance variable *radius* has specified as a floating point value. A PatientRecord object that stores hospital information about a patient would obviously return a lot more data in an organized format than the Circle object. For testing purposes, we override toString() in many of our classes to provide a simpler way to represent the Class instead of calling all of its methods to display the instance variables.

For Next Week:
Review your **Card** and **Deck** classes.
Resolve your **Shuffle** functions if those are not complete and/or understood.
Think about the game logic for BlackJack, Poker, and other common playing card games. If you don't know these games, go play them!