AP Computer Science A - Ardent Academy Fall 2018
Class Notes 4

Today's Agenda:
1. Student questions.
2. Review and code loops: while, for
3. RPS modification - solution
4. Alternative loops: do-while and for-each
5. Data Structures: Arrays and ArrayList

Loops

A **while** loop is best used when you do not know how many times you plan to *iterate* your block of code, but are dependent on the condition being true to continue looping.

A **for** loop is best used when you want to *iterate* a specific number of times.

Increment/Decrement

C style languages can increment and decrement using ++ and --.

These can appear as a post-operator (post-op):

```
counter++;
counter--;
```

They can also appear as a pre-operator (pre-op):

```
++counter;
--counter;
```

The difference is in WHEN the increment or decrement happens, before or after the statement is executed.

This behavior is part of the reason why languages like Python do not allow the use of this kind of operator, and rely solely on += and-= for increment and decrement operations.

Programming Exercise

1. Write a while loop that prints all of the numbers divisible by 7 between 1 and 100 inclusive in ascending order.

2. Write a for loop that does the equivalent.

3. Write a for loop that prints all of the numbers divisible by 3 between 100 and 1 inclusive in descending order.

4. Write a while loop that does the equivalent.

Global Variables

A global variable is a variable which can be read anywhere in a computer program. They are usually declared at the very beginning (and top) of your code.

While this gives us an incredibly amount of access and flexibility to use that variable, this isn't really a good way to program. A global variable can be seen and/or changed by anything in the program...this can be dangerous when we have large software systems. In our code here, it's probably not that big of an issue, but you should not always employ this practice of create variables that everyone can always see...more on this later as we move into Procedural and Object Oriented Programming.

Alternative Versions of Loops

While has a second form called the **do-while**:

```
do
{
    // code block
}
while( condition );
```

In this version, the block of code is executed *first* before the condition for the loop is checked. This is useful where you want the code block to take place at least one time before we start looping.

For has an alternate form called **for-each**. It has a very special purpose for Strings and Arrays.

```
String name = "Bob";

// "for each char letter in the String name, print
for( char letter : name )
{
    System.out.println( letter );
}
```

output:
B
o
b

Arrays

So far, we've stored information using single variables. But what would happen if we had to store hundreds of names, or hundreds of numbers? It would be painful to have to declare hundreds of unique variable names to store all of them.

The solution to this is called a *data structure*, which can hold multiple elements/variables. One that is built-in to Java is called the **array**.

An **array** can hold multiple variables of all the same type. It is declared like this:

```
// declare an array of integers
int[] nums;

// declare and instantiate an array of ten integers
int[] nums = new int[ 10 ];

// declare, instantiate, and prepopulate
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Arrays can be declared using any primitive or abstract data type. You can put integers, floating point, Strings, etc. inside an array. Once declared though, you cannot change any array's type, nor can you change the number of elements that it can store.

In this way, we see that arrays are **finite** in size, and that cannot be changed after they are instantiated. This is the fundamental limitation of standard arrays in Java.

Accessing items in an array is made possible by referencing the **index position** of data inside the array. The index position is a unique address for a piece of information. It begins counting at 0:

```
String[] names = { "Bob", "Jennifer", "Chuck", "La-a" };
index position:     0      1           2        3

System.out.println( names[ 2 ] );
output: Chuck

System.out.println( names[ 0 ] );
output: Bob

System.out.println( names[ 4 ] );
output: ArrayIndexOutOfBounds Error
```

```
// for loops love arrays!

int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

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

output:
1 2 3 4 5 6 7 8 9 10

Each array has a variable in it called *length* which is an integer that represents the total number of elements that the array can store. We can use this variable in the condition for a for loop to iterate over every single element inside it. This sequential operation is what makes for loops so uniquely suited for traversing the data inside of an array, in a process that we call **linear search**.

Here's an example of a linear search on an abstract data type called *MedicalRecord*:

```
MedicalRecord[] pRecords = new MedicalRecord[100000000];
// assume pRecords is now populated 100 million med records

// linear search for "Bruce"
for( int i = 0; i < pRecords.length; i++ )
{
    if( pRecords[ i ].getFirstName() == "Bruce" )
    {
        System.out.println( "Possible match." );
        System.out.println( "Location: " + i );
    }
}
```

The one attribute of arrays that limits their power is that they are **finite** in size, and that cannot change after they have been instantiated. Imagine what would happen with our medical records here if the number of records exceeds 100 million!?

To deal with this problem, we need a kind of array that behaves like a **list**, which is *dynamic* and will shrink or expand to accommodate the amount of data that it has. Java doesn't have one that is built-in, so it is provided using a library, called **ArrayList.**

```
import java.util.ArrayList;

// new arraylist of strings
ArrayList<String> names = new ArrayList<String>();

names.add( "Bob" );
names.add( "Ted" );
names.add( "Sally" );

names [ "Bob", "Ted", "Sally" ]
index:   0      1      2

names.add( 0, "Pikachu" );

names [ "Pikachu", "Bob", "Ted", "Sally" ]
index:   0          1      2      3

names.remove( 1 );

names [ "Pikachu", "Ted", "Sally" ]
index:   0          1      2

names.remove( "Ted" );

names [ "Pikachu", "Sally" ]
index:   0          1

System.out.println( names.size() );

output: 2
```

ArrayLists have one limitation...they can really only store Abstract Data Types. In order to get around this problem, they have two special types that they use to store integers and doubles, called *Integer* and *Double*. These special types are called **wrapper classes**. The Java system automatically takes primitive int or double values and puts them into these special types in a process called **autoboxing**. They will also decode them from the special type back into a primitive value in a process called **autounboxing**. You personally don't need to worry about it, since it does it automatically, but do understand that this built-in behavior is what allows ArrayList to deal with integers and doubles.

```
ArrayList<Integer> nums = new ArrayList<Integer>();
nums.add( 3 );
nums.add( 4 );
nums.add( 6 );

ArrayList<Double> nums = new ArrayList<Double>();
nums.add( 3.14159 );
nums.add( 4.2 );
nums.add( 66.000064321 );
```

To know everything that an ArrayList is capable of doing, look at the online language reference for it here:

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

--

Strings

Strings store "strings of characters" (of the char data type), and are used by all modern computer systems to store *text*. Text data is the primary form of data that humans use to communicate using computers.

Manipulating text is a key feature of computing in fields like language analysis, linguistics, security (message filtering).

Strings are **immutable**. This means that after a string has been created, the individual characters within each string cannot be modified directly. This is done for security purposes, so that when a username or password is transmitted over a network, malicious software cannot insert itself into a string and thus cause a computer to execute invalid code that could compromise the system. We call this a "man-in-the-middle" attack in computer security.

In order to edit individual letters in a string, we need to take advantage of a function that strings have:

```
String name = "Bob";
char[] letters = name.toCharArray();

char[] newName = new char[ 5 ];

for( int i = 0; i < letters.length; i++ )
{
    newName[ i ] = letters[ i ];
}

newName[ 3 ] = 'b';
newName[ 4 ] = 'y';

String sNewName = String.valueOf( newName );
System.out.println( sNewName );
```

One really important function that String can perform is to look at just a portion of something inside a larger string. This is part of searching, especially if we are looking for a specific word inside of a sentence.

```
String sentence = "I love dogs.";
String animal = sentence.substring( 7, 11 );
System.out.println( animal );
```

output:
dogs

The substring() function accepts two variables, the first is the starting index position (inclusive) of the substring that you want, and the second is the ending index (exclusive).

```
String sentence = "I love dogs.";
String animal = sentence.substring( 7, 10 );
System.out.println( animal );
```

output: dog

Programming Project - WordGuess

You might be familiar with a children's word guessing game called Hangman. In this game, a secret word is presented to the players using dashes or underscores representing the number of letters that they must guess. Each player tries to guess individual letters. The game ends when all of the letters have been guessed, or the players run out of guesses, represented by drawn parts of a hanged man. While this game was quite popular in the 80's in schools, the nature of the "hangman" has caused it to diminish in popularity.

We are going to write a program that allows a human player to play a word guessing game against the computer.

Algorithm
1. Create a list of secret words. Use an *array* or an *ArrayList*.
2. Randomly select a word from the list of secret words.
3. Make a dashed word that has the same number of letters as the secret word, but displays as - - - - -.
4. outside LOOP until the user has guessed all of the letters
5. show the user the current dashed word
6. prompt the user to guess a letter
7. inside LOOP through the secret word; if the guess matches any of the letters, replace the dash with the letter

Use this **algorithm** to create your code comments.
When you **implement** the program, go in order and do not skip steps!
Test your program at every step!

Q: What is the difference between an array and an ArrayList?

An *array* is a data type that is one of the default types available in Java. You don't need to import anything to use an array. An array can store any type of data, primitive or abstract. An array is finite in size.

```
String[] names = { "Bob", "Sally", "Ted" };
System.out.println( names.length );
```

An *ArrayList* is a special data type that you have to import to use, from the *java.util* library. It is a dynamic implementation of an array, which scales in size to fit however little or a lot of data that you put into it. It can store only abstract data types, but has built-in features to handle primitives.

```
import java.util.ArrayList;
ArrayList<String> names = new ArrayList<String>();
names.add( "Bob" );
names.add( "Sally" );
names.add( "Ted" );
System.out.println( names.size() );
```

Stopping a loop

There is a command that we used with the switch statement that we can use here that will stop a loop immediately:

```
break;
```

Use it at your own peril.