

AP COMPUTER SCIENCE A

This guide was created through a multitude of sources and personal notes. Sources Below
From Simple Studies, <https://simplestudies.edublogs.org> & @simplestudiesinc on Instagram

Unit 1: Primitive Types

Print statements and commenting:

The System.out.print and System.out.println methods are used to send output for display on the screen.

System.out.println moves cursor to new line after printing.

Ex: System.out.println("Hello");

System.out.print("World");

Output: Hello

World

System.out.print does not move the cursor to a new line after printing.

Ex: System.out.print("Hello");

System.out.print("World");

Output: Hello World

To comment a single line use //

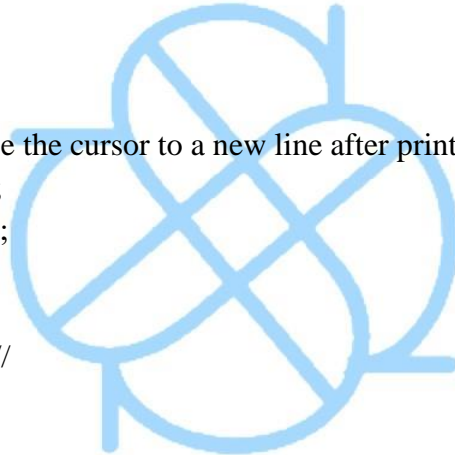
Ex: //This is a return method

To comment multiple lines use */* comment */*

Ex: */* comments*

are ignored by computer

**/*



Data Types:

Type:	Description:	Examples:
int	integers	2, 5 , -3
double	decimals	2.45 , 3.14, -8.1
boolean	True or false	True, false

Operators:

+	addition
-	subtraction
*	multiplication
/	division
%	modulus

Compound operators and increment and decrement:

increment	x++	x+=1	x= x + 1
decrement	x--	x-=1	x= x - 1

Corresponding operator for arithmetic operators:

Compound operator:	Other way to write it:	Example:
+=	x= x + 4	x+=4
-=	x= x - 4	x-=4
*=	x= x * 4	x*=4
/=	x= x / 4	x/=4
%=	x= x % 4	x%=4

Casting:

Values of a certain type can only be stored in a variable of that type. Casting is changing a bigger data type to a smaller data type.

You must cast: double to integer if casting.

Ex: double d = 5.25;

```
int i = (int) d;
```

To turn a decimal to int :

Ex: int x = (int) 17.654 // Put the (data type) in front of the value to change it

Example of mathematical operation:

```
int x = 10;
```

```
double y = 2 * x + 3; // y will store the value 23.0
```

Unit 2: Using Objects

Objects:

An object is created from a class by calling the class constructor along with the *new* keyword.

The name of a constructor is the same as the name of the class it belongs to.

If it is followed by a (possibly empty) list of values then those values are called parameters.

The signature of a constructor consists of the name of the constructor along with the list of types of parameters that it expects. When calling the constructor, the parameter list provided must match the signature.

Ex: The *Rectangle* class from the Java standard library contains, among others, the following two constructors:

```
Rectangle(int width, int height) // first constructor
```

and...

```
Rectangle(int x, int y, int width, int height) // second constructor
```

Valid constructor calls:

```
new Rectangle(3, 6) // calls the first constructor
```

```
new Rectangle(-1, 7, 3, 4) // calls the second constructor
```

Not valid constructor calls:

```
new Rectangle(7.45, 1) // invalid since the first parameter is double
```

```
new Rectangle(1, 7, 3) // invalid since it has three parameters
```

Creating and Storing Objects (Instantiation):

Instantiation for rectangle class:

```
Rectangle myRectangle = new Rectangle(5, 6);
```

myRectangle is a reference variable that refers to an Rectangle object.

Calling a Void Method:

A method is a storage location for related program statements. When called, a method usually performs a specific task.

Ex:

```
Public class Bag
```

```
{
```

```
    Public void sand()
```

```
    {
```

```
        System.out.println("Hi");
```

```
    }
```

```
}
```

Methods are defined inside of a class and you can define as many methods as needed. Method sand does not return a value! Once you have instantiated an object you can call all the methods contained in the class. Void methods do not return a value.

Calling Non-Void Methods:

When a method is not void, it has a return type, therefore the method returns a value.

Ex:

```
Public class Train
```

```
{
```

```
    Public int rides ( int ride1, int ride 2)
```

```
    {
```

```
        return ride1 * ride2;
```

```
    }
```

```
}
```

Return methods are defined inside of a class in the same way you define void methods.

```
Train comp = new Train();
```

```
int store = comp.rides(3, 6);
```

String Objects: Concatenation, Literals, and More:

A string is a group of characters. Strings are enclosed in “ ”. Keep in mind even numerals if enclosed in quotes are considered strings. Strings can be combined or concatenated. The concatenation of strings is shown below:

```
String one= "Hello";
```

```
String two= "World";
```

```
String concat = one+ " " + two;
```

```
System.out.print(concat); // Prints Hello World
```

Escape Sequences:	Meaning:
\"	Prints “
\\	Prints \
\n	New line

String Method:	Use:
substring(x,y)	Returns a section of the string from x to y not including y
substring(x)	returns a section of the string from x to length-1
charAt(x)	Returns the char at spot x
length()	Returns number of chars
indexOf(str)	Returns location of String str in the string, searching from spot 0 to spot length-1
indexOf(ch)	Returns the location of char ch in the string, searching from spot 0 to spot length-1
lastIndexOf(str)	Return the location of String str in the string, searching from spot length-1 to spot 0
lastIndexOf(ch)	Returns the location of char ch in the string, searching from spot length-1 to spot 0
equals(s)	Checks if this string has same chars as s

compareTo(s)	Compares this string and s for >,<, and ==
--------------	--

String Indexes:

0	1	2	3	4
L	e	m	o	n

L is at index or spot 0... etc.

Length of this string is 5.

Wrapper Classes: Integer and Double:

Sometimes it is more convenient to work with objects rather than primitive values because of this, Java provides the wrapper classes Integer and Double. Each of these classes has a constructor that accepts a primitive value of the appropriate type, and a method that returns the primitive value.

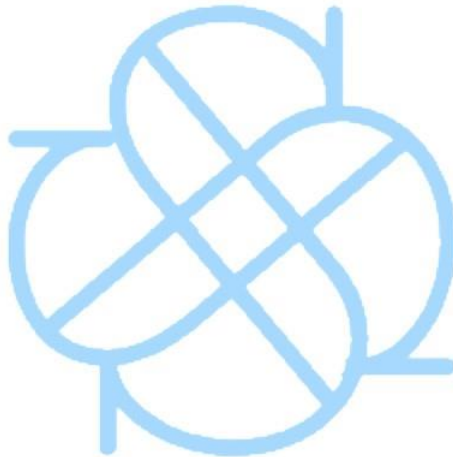
The Integer class also provides static fields that represent the maximum and minimum values that can be represented by an int.

Integer:	Double:
Integer(int value) Constructs an Integer object with the specified value	Double(double value) Constructs a Double object with the specified value
int intValue() Returns the stored int value	double doubleValue() Returns the stored double value
Integer.MAX_VALUE The maximum value that can be represented by an int	
Integer.MIN_VALUE The minimum value that can be represented by an int.	

The Math Class:

Math class only contains static methods.

Method:	Description:
<code>int abs(int x)</code>	Returns the absolute value of x
<code>double abs(double x)</code>	Returns the absolute value of x
<code>double pow(double b, double e)</code>	Returns b^e
<code>double sqrt(double x)</code>	Returns the square root of x
<code>double random()</code>	Returns a value in the interval $[0,1)$



Unit 3: Boolean Expressions and if Statements

Boolean Expressions:

A Boolean value is either true or false. These literals can be used, however Boolean expressions can also be created by using relational operators(<, <=, >, >=, ==, and !=) with primitive data types.

If Statement and Control Flow:

Control flow statements are used when a program needs to make decisions based on its current state. An if statement allows the program to either execute or skip a section of code based on whether a Boolean expression is true or false.

Example:

```
if (expression)
{ // one or more statements }
```

If expression evaluates to true, the statements between the { } are executed. Otherwise, the statements in the body are skipped.

If-Else Statements:

An if-else statement can be created if an else clause is added. If expression evaluates to true, the statements between the { } are executed, if false then the statements between the else { } are executed.

```
if (expression)
{ // one or more statements }
else
{ // one or more statements }
```

If Else If Statements:

For multiple possibilities, an if can be followed by one or more else if clauses.

```
if (expression1)
{ // statements1 }
else if (expression2)
{ // statements2 }
else
{ // statements3 }
```

Compound Boolean Expressions:

Compound Boolean expressions can be formed using logical operators && (and), || (or), and !

&&	A && B is true only when A and B are both true
	A B is true when at least one of A or B is true
!	!A is true only when A is false

Equivalent Boolean Expressions:

Two Boolean expressions are equivalent if they evaluate to the same truth value for all possible values of their variables, ie; $\neg(A \ \&\& \ B)$ is equivalent to $\neg A \ || \ \neg B$

Comparing Objects:

Reference variables store references to objects, rather than the objects themselves, because of this, when objects are compared using == and !=, it is only the references that are being compared, not the contents of the actual objects. These operators only check if two references are aliases of each other. Additionally, == and != can be used to check whether a reference variable is null. Only use == and != if you are either comparing primitive values or checking to see if a reference variable is null.

Comparing objects themselves for equality can only be achieved if the class provides an equals method, which exists for String.

Unit 4: Iteration

While Loops:

The while statement allows a code block to repeat as long as a condition is true.

```
while (expression)
{ // one or more statements }
```

If expression is true the code will run, after it runs expression is checked again and if true will run, this continues until expression is false. If expression is false the first time the body of code will not run. When writing a while loop make sure that the condition becomes false or else the code will execute infinitely.

For Loops:

A for loop uses a variable to count the iterations of a loop.

```
for (initialization; expression; increment)
{ // statements }
```

When the loop is first encountered, the initialization statement is executed. Then expression is evaluated. If it is true, the loop body is executed. After execution, increment is executed, and expression is evaluated again. This continues until expression is false, at which point the loop is done.

Developing Algorithms and Using Strings:

Loops can be used to process Strings; the indexOf and substring methods are a few ways strings can be processed.

```
for (int i = 0; i < myStr.length() - k + 1; i++)
{ String subs = myStr.substring(i, i + k);
// do something with subs }
```

Here “i” is used to keep track of the index in a string. In each iteration a substring of length k is retrieved starting at index i.

Nested Iteration:

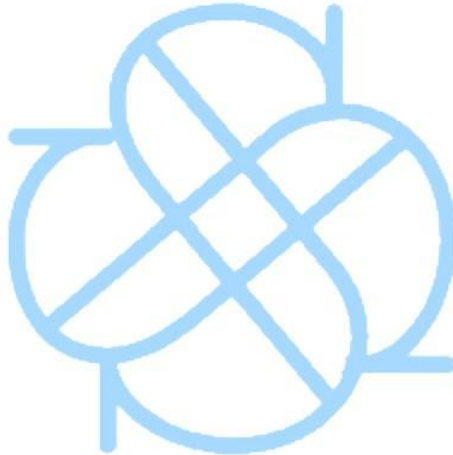
A nested loop is when a loop is put in the body of another loop. Nested loops are commonly used in more complex string and array algorithms.

```
for (int x = 5; x >= 1; x--)  
{ for (int y = 0; y < x; y++)  
  { System.out.print(y + " "); }  
  System.out.println(); }
```

Informal Code Analysis:

What makes a good algorithm:

- correctness, does the code work as intended
- efficiency, as little unnecessary or overcomplicated code as possible
- easily understandable by others/ will someone be able to understand what the code is supposed to do



Unit 5: Writing Classes

Anatomy of a Class:

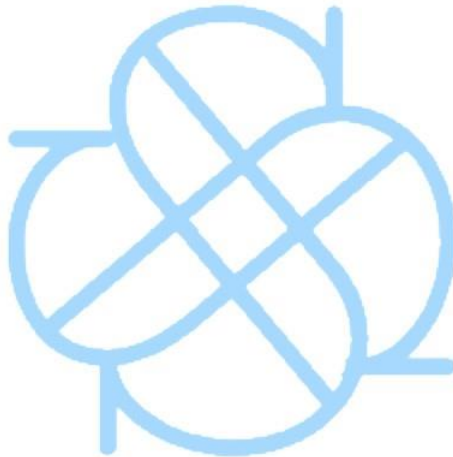
All members with public access can be accessed or modified inside or outside of the class where they are defined.

When you need many methods to have access to the same variable, you make that variable an instance variable. The scope of an instance variable is the entire class where the variable is defined.

Ex:

Public class Run

```
{  
    private int one, two;  
    private int answer;  
  
    public void jog()  
    {  
        answer = one + two;  
    }  
  
    Public int getAnswer()  
    {  
        return answer;  
    }  
}
```



Always keep in mind:

- All instance variables should be private
- All constructors should be public
- Methods may be public or private

Constructors:

They are very similar to methods. They also have the same name as the class and have no return type – no void, int, etc. Make sure to initialize all instance variables.

Steps to create a constructor heading:

1. public (access type)
 2. Name (always has same name as class)
 3. () and must have own set of curly braces
- There are default constructors and initialization constructors.

Default constructors have no parameter list. When a default constructor is called, all instance variables / data fields are set to a zero value. If no constructors are provided for a class, Java will provide a default constructor that will initialize all data to a zero value.

Initialization constructors have a parameter list and will receive parameters when called. The number of parameters and types passed in must match up with the parameter list following the method name.

Documentation with Comments:

In addition to single and multi-line comments, Java has a third comment syntax, which generates Javadoc documentation. These comments are enclosed by `/** comment */`, and are used to document the description, purpose, and conditions associated with a class, instance variable, constructor, or method.

A precondition is a condition that must be true immediately prior to a method being called. If it is not true, the method should not be expected to work as described and may cause an error to occur.

Accessor Methods:

Accessor methods are methods that retrieve or grant access to the properties of an object, but do not make any changes.

Ex:

```
public String printA()
{
    return sideA; //sideA is an instance variable
}
```

Mutator Methods:

Mutator/Modifier methods are methods that change the properties of an object.

Ex:

```
public void setSides(int a, int b, int c)
{
    sideA=a;
    sideB=b;
    sideC=c;
}
```

Writing Methods:

A method is a block of code that exists within a class and has access to the instance variables.

The syntax for writing a method is as shown:

```
visibility returnType methodName(parameters)
{ // method body }
```

Visibility can be either public or private. Most methods are public, but some situations might use private to keep the accessibility of a method limited to its class. The return type specifies what type of value, the method will return. If the method will not return any value, the keyword void is used in place of the return type.

Remember: A method is called an accessor if it returns something, but does not modify data. To return a value from a method, a return statement is used.

Also note: A return statement will immediately terminate the execution of the method. Any code that follows it will never be reached, and if it is within a loop there will be no more iterations.

Static Variables and Methods:

For example:

```
AplusBug cs = new AplusBug();
AplusBug dude = new AplusBug();
```

A reference variable is used to store the location of an Object. In most situations, a reference stores the actual memory address of an Object.

In the example cs and dude store the location / memory address of two new AplusBug Objects.

A variable is a storage location for a specific type of value.

Static variables and methods are associated with a class, rather than with instances of the class, and are declared by including the static keyword.

Ex:

```
private static int count;
public static double getValue() { ... }
```

Static methods can be either public or private, but they only have access to static variables that cannot read or change any instance variables. In particular, they do not have access to the *this* keyword (We will talk about *this* keyword later in this unit).

Scope and Access:

In variable scope when a variable is defined within a set of braces, that variable can only be accessed inside those braces.

As shown below:

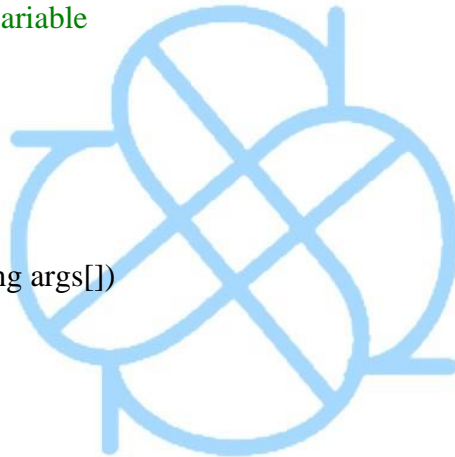
```
{  
    int fun = 99;  
}
```

The variable has a scope limited to those braces.

When you need only one method to have access to a variable, you should make that variable a local variable. The scope of a local variable is limited to the method where it is defined.

Example including instance and local variables:

```
public class Can  
{  
    private int soup;    //instance variable  
  
    public void food() {  
        int soup= 99;    //local variable  
    }  
  
    public void print() {  
        System.out.println(soup);  
    }  
  
    public static void main(String args[])  
    {  
        Can test = new Can();  
        test.change();  
        test.print();  
    }  
}
```



This Keyword:

The keyword *this* can be used in a class to refer to the current calling object.

For example, in the following Class Person, when we create an object p1 and call the constructor or p1.setEmail(), the word *this* refers to p1. And when we make the same method calls with object p2, *this* refers to p2.

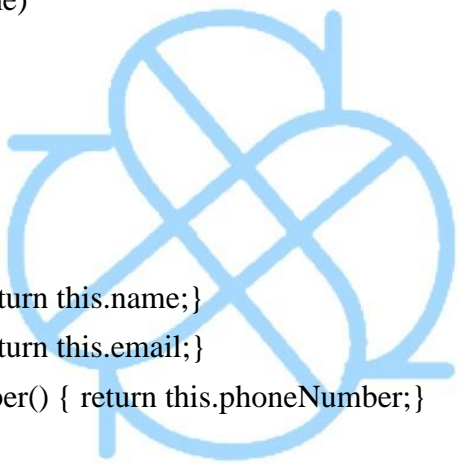
```
public class Person
{
    // instance variables
    private String name;
    private String email;
    private String phoneNumber;

    // constructor
    public Person(String theName)
    {
        this.name = theName;
    }

    // accessor methods - getters
    public String getName() { return this.name;}
    public String getEmail() { return this.email;}
    public String getPhoneNumber() { return this.phoneNumber;}

    // mutator methods - setters
    public void setName(String theName) { this.name = theName;}
    public void setEmail(String theEmail) {this.email = theEmail;}
    public void setPhoneNumber(String thePhoneNumber) { this.phoneNumber =
thePhoneNumber;}
    public String toString()
    {
        return this.name + " " + this.email + " " + this.phoneNumber;
    }

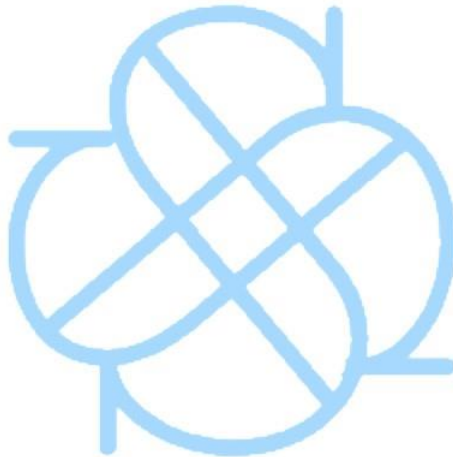
    // main method for testing
    public static void main(String[] args)
```



```
{  
    Person p1 = new Person("Sana");  
    System.out.println(p1);  
    Person p2 = new Person("Jean");  
    p2.setEmail("jean@gmail.com");  
    p2.setPhoneNumber("404 899-9955");  
    System.out.println(p2);  
}  
}
```

EXAMPLE CODE ABOVE AND THIS KEYWORD DEFINITION IS FROM:

<https://runestone.academy/runestone/books/published/csawesome/Unit5-Writing-Classes/topic-5-9-this.html>



Unit 6: Arrays

Array Creation and Access:

An array is a collection of boxes / spots / items that all store the same type of value.

Each spot in the array stores a value of the same type.

Ex:

`int[] array = new int[10];` array can store 10 integers. This array is basically a collection of 10 integer variables.

Spot 0 stores the 1st integer, spot 1 stores the 2nd integer, and so on.

0	1	2	3	4	5	6	7	8	9	Index positions in green
0	0	0	0	0	0	0	0	0	0	

Above is a visual representation of the array that is created by `int[] array = new int[10];`

Note that: Once an array object has been instantiated, the size may never change. To increase or decrease the size, a new array would need to be instantiated and all old values copied.

Also note: The size of an array object can never change. Arrays do not have methods that allow for the removal or addition of items. In order to add or remove items, a new array would be instantiated and all old values copied to the new array.

To initialize an array with values here is an example:

`int[] aplus = {2,7,8,234,745};` // index positions are in green

And a visual representation of the array:

0	1	2	3	4
2	7	8	234	745

Indexes for arrays will always be integers and will always start with 0.

Setting array values:

`int[] nums = new int[10];`

`nums[0] = 231;`

`nums[4] = 756;`

`nums[2] = 123;`

New Array:

231	0	123	0	756	0	0	0	0	0
-----	---	-----	---	-----	---	---	---	---	---

```
System.out.println(nums[0]); //prints value at index position 0
System.out.println(nums[1]); //prints value at index position 1
System.out.println(nums[4]); //prints value at index position 4
System.out.println(nums[4/2]); //prints value at index position 2
```

Accessing Array Values- If statements can be used to access and check specific values in the array. Sometimes it is necessary to check a value in a specific location against another value in the array at a different location. The if statement below is comparing the value at spot 1 to the value at spot 2.

Ex:

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

```
if ( nums[1] == nums[2] )
    out.println( "aplus" );
else
    out.println( "comp" );
```

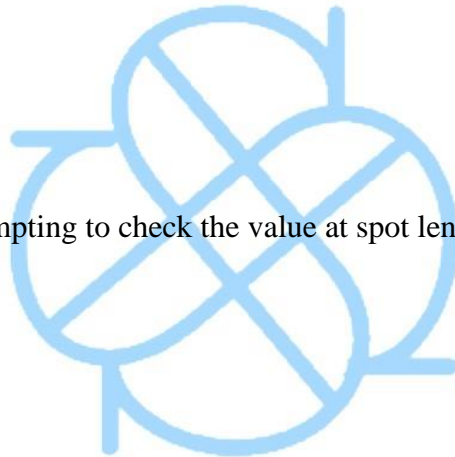
The if statement below is attempting to check the value at spot length to see if it is 7.

```
int[] nums = {1,2,3,4,5,6,2};
```

```
int last = nums.length;
if( nums[last] == 7 )
    out.println( "aplus" );
else
    out.println( "comp" );
```

The if statement is out of bounds because the last value in the array is stored at position length-1 not length.

An `ArrayIndexOutOfBoundsException` is thrown.



Traversing Arrays + Enhanced for loops (for- each loops):

Using loops to print all spots in an array is a necessary approach. Array lengths could change with different input values, it is good to use a for loop based on length. If length changes, the loop will change accordingly.

How the loop works:

The loop variable will start at 0 and go up to the array length.

The loop variable will be used to access each [spot] in the array.

Ex:

```
int[] nums = {3,2,5,1,0,6};
```

```
for(int spot=0; spot<nums.length; spot++)  
{  
    System.out.println(nums[spot]);  
}
```

The output would be:

3
2
5
1
0
6

Another way:

The for each loop is a great thing to use when accessing array values if a spot/index variable is not needed.

The for each loop below accesses all values in nums and prints each one.

Each time the loop iterates, the next value from nums is pasted into the item.

The for each loop will iterate as long as the structure it is connected to has values.

```
int[] nums = {1,2,3,4,5,6,7};  
for(int item : nums)  
{  
    out.print(item + " ");  
}
```

Output is 1 2 3 4 5 6 7

You can also use for loops to change values in the array as shown below:

```
int[] nums = new int[6];
for(int spot=0; spot<nums.length; spot++)
{
    nums[spot] = spot*4;
}
```

This would output: (Index positions are in green)

0	1	2	3	4	5
0	4	8	12	16	20

You can also traverse String arrays as shown below:

```
String[] wrds = {"cat","pig","dog"};
for(String item : wrds)
{
    out.println(item);
}
```

This would output: (Index positions are in green)

cat	pig	dog
-----	-----	-----

Developing Algorithms Using Arrays:

There are several standard algorithms that traverse arrays and process the elements within:

- Find a maximum or minimum in an array
- Compute the sum or mean of the values in an array
- Determine whether some or all of the elements in an array have a certain property
- Count the number of elements in an array satisfying a certain condition
- Access all pairs of consecutive elements
- Shift or rotate all elements in an array to the right or left

Be familiar with these !!!

THIS IS FROM:

http://support.ebsco.com/LEX/AP-Computer-Science-A_Study-Guide.pdf

Unit 7: ArrayLists

Introduction to ArrayList:

ArrayList is a class that houses an array. An ArrayList can store any type. All ArrayLists store the first reference at spot / index position 0.

An ArrayList is different from an array because its size is not set at creation and can change.

To create an ArrayList, call its default constructor:

```
ArrayList house= new ArrayList();
```

More examples:

```
ArrayList<String> words;
```

```
words = new ArrayList<String>(); //can only store string references
```

```
List<Double> decNums;
```

```
decNums = new ArrayList<Double>(); //can only store double references
```

ArrayList Methods:

Name:	Use:
add(item)	Adds item to the end of the list
add(spot, item)	Adds item at spot- shifts items up->
set(spot, item)	Put item at spot $z[\text{spot}] = \text{item}$
get(spot)	Returns the item at spot return $z[\text{spot}]$
size()	Returns the # of items in the list
remove(int spot)	Remove item at spot from the list

These are the most important and commonly used methods when using ArrayLists.

Traversing ArrayLists:

Below there is an example of a standard for loop. Note that it says `.size()` . This returns the number of elements.

Ex:

```
for (int i=0; i<ray.size(); i++)  
{  
    out.println(ray.get(i));  
}
```

Another example of a standard for loop:

```
List<Integer> ray;
```

```
ray = new ArrayList<Integer>();
```

```
ray.add(23);  
ray.add(11);
```

```
for( int i = 0; i < ray.size(); i++){  
    out.println( ray.get( i ) );  
}
```

The output:

```
23  
11
```

An example below is of a standard for each loop:

```
for ( int item : bunchOfNums )  
{  
    out.println( item );  
}
```

The for each loop will access each reference in the list, starting with the first and ending with the last.

Another example of a for each loop:

```
List<Integer> ray;  
ray = new ArrayList<Integer>();
```

```
ray.add(23);  
ray.add(11);  
ray.add(53);
```

```
for(int num : ray){  
    out.println(num);  
}
```

This will output:

```
23  
11  
53
```

Some extra examples using ArrayList Methods:

The add method with Strings:

```
ArrayList<String> vals;  
vals = new ArrayList<String>();  
vals.add("aplus");  
vals.add("rocks");  
vals.add(0, "comp");  
vals.add(1, "sci");  
out.println(vals);
```

Output: [comp, sci, aplus, rocks]

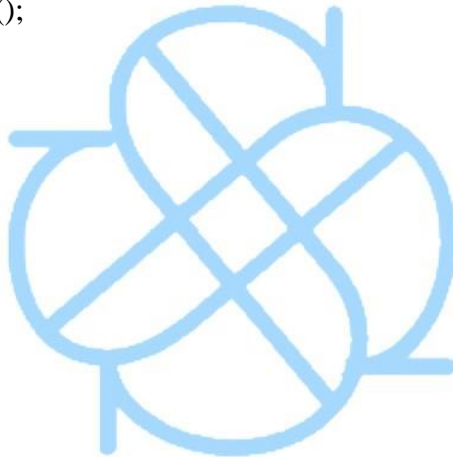
The set method with integers:

```
ArrayList<Integer> ray;  
ray = new ArrayList<Integer>();  
ray.add(23);  
ray.add(11);  
ray.set(1,5);  
ray.add(4);  
ray.set(0,7);  
ray.add(53);  
out.println(ray);
```

Output: [7, 5, 4, 53]

```
List<Integer> ray;  
ray = new ArrayList<Integer>();  
ray.add(23);  
ray.add(11);  
ray.add(12);  
ray.add(65);
```

```
out.println(ray.get(3));  
out.println(ray.get(4));
```



Output:

65

Runtime Exception Error (Because there is no 4th index position)

```
ArrayList<String> ray;  
ray = new ArrayList<String>();
```

```
ray.add("a");  
ray.add("b");  
ray.remove(0);  
ray.add("c");  
ray.add("d");  
ray.remove(0);  
out.println(ray);
```

Output: [c, d]

ALL ABOVE EXAMPLES FROM WESTLAKE POWER POINTS!!

Developing Algorithms Using ArrayLists:

Note when developing Algorithms:

- The length of an ArrayList is obtained from its size() method.
- Accessing the element at position i uses the .get(i) method call.
- Since the size of an ArrayList can change by adding and removing elements, the algorithms previously discussed can be extended to perform these actions.
- The size of an ArrayList cannot be changed within the body of an enhanced for loop; this will result in an error.

Searching and Sorting:

Selection sort is pretty effective for small lists, but not so effective when used on large lists.

Selection sort consists of two loops:

The outer loops run based on the number of items in the list.

The inner loop runs to find the items that need to be moved.

The inner loop either locates the spot with the smallest value or the spot with the largest value.

After the inner loop completes, a swap may occur if needed.

At most, selection sort will make one swap per pass.

A pass is one complete execution of the inner loop.

Selection sort is a “search and swap” algorithm.

Example 1 of Sort Algorithm:

```
void selectionSort( ArrayList<Integer> stuff )
{
    for(int i=0; i< stuff.size()-1; i++){
        int min = i;
        for(int j = i+1; j< stuff.size(); j++)
        {
            if( stuff.get(j) < stuff.get(min) )
                min = j;    //find location of smallest
        }
        if( min != i ) {
            int temp = stuff.get(min);
            stuff.set( min, stuff.get(i) );
            stuff.set( i, temp);    //put smallest in pos i
        }
    }
}
```

```
}
```

Example 2 of Sort Algorithm:

```
void insertionSort( ArrayList<Integer> stuff)
```

```
{
    for (int i=1; i< stuff.size(); ++i)
    {
        int val = stuff.get(i);
        int j=i;
        while( j>0&&val<stuff.get(j-1) ){
            stuff.set( j, stuff.get(j-1) );
            j--;
        }
        stuff.set(j,val);
    }
}
```

Collections Class Methods:

Name:	Use:
sort(x)	Puts all items in x in ascending order
binarySearch(x, y)	Checks x for the location of y
fill (x, y)	Fills all spots in x with value y
rotate(x, y)	Shifts items in x left or right y locations
reverse(x)	Reverses the order of the items in x

Example using Collections.sort():

```
ArrayList<Integer> ray;
```

```
ray = new ArrayList<Integer>();
```

```
ray.add(23);
```

```
ray.add(11);
```

```
ray.add(66);
```

```
ray.add(53);
```

```
Collections.sort(ray);
```

```
out.println(ray);
```

```
out.println(Collections.binarySearch(ray,677));
```

```
out.println(Collections.binarySearch(ray,66));
```

The Output: [11, 23, 53, 66]

-5
3

Example using Collections.rotate():

```
ArrayList<Integer> ray;  
ray = ArrayList<Integer>();
```

```
ray.add(23);  
ray.add(11);  
ray.add(53);  
out.println(ray);  
rotate(ray,2);  
out.println(ray);  
rotate(ray,2);  
reverse(ray);  
out.println(ray);
```

The Output: [23, 11, 53]
 [11, 53, 23]
 [11, 23, 53]

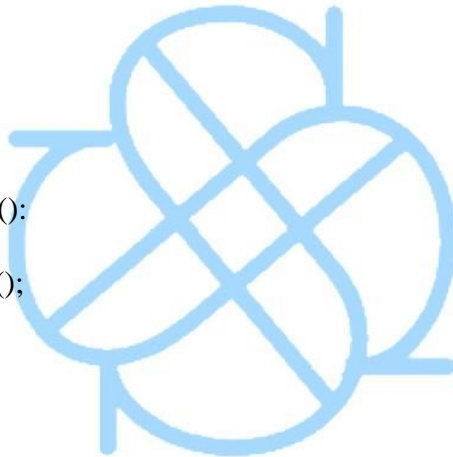
Example using Collections.fill():

```
ArrayList<Integer> ray;  
ray = new ArrayList<Integer>();  
ray.add(0);  
ray.add(0);  
ray.add(0);  
out.println(ray);
```

```
Collections.fill(ray,33);  
out.println(ray);
```

The Output: [0, 0, 0]
 [33, 33, 33]

ALL EXAMPLES FROM WESTLAKE NOTES!!



Unit 8: 2D Arrays

2D Arrays:

A 2D Array or Matrix is A two-dimensional array is a one-dimensional array of one-dimensional arrays. A matrix has rows and columns.

Ex: A spreadsheet is a matrix.

What does this matrix look like? Shown below!

`int[][] rayORays = new int[3][3];` (Index positions are green and also go vertically)

0	1	2
0	0	0
0	0	0
0	0	0

What does this matrix look like? Shown below!

`int[][] mat = {{6, 9, 2}, {5, 3, 4, 6}};`

6	9	2	
5	3	4	6

More Examples:

`String[][] words = new String[4][4];`
//words is filled with 16 nulls

`double[][] dMat = new double[3][3];`
//dMat is filled with 9 0.0s

Reminder: To create a 2D Array jut extend the notation for a 1D array.

For example, in the array below *type* is the type of value being stored, *rows* is the number of rows, and *cols* is the number of columns.

`type[][] name = new type[rows][cols];`

A 2D array can be initialized with items that are each initializer lists for a 1D array:

```
int[][] values = new int[][] {{1, 2, 3}, {4, 5, 6}};
```

If arr is a 2D array, the expression arr[r] refers to the element at position r in arr. But this element is itself a 1D array, so it can be further indexed with the expression arr[r][c].

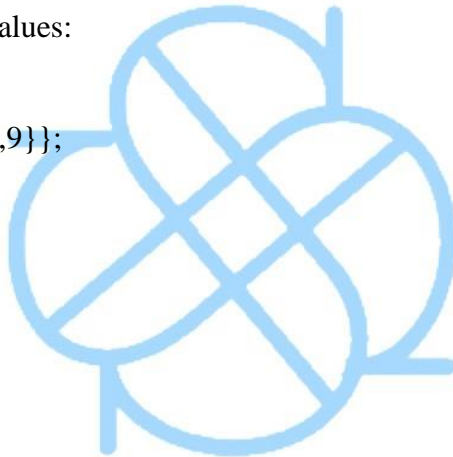
The two subscripts should be thought of as the row and column positions in the rectangular array. The row and column indices both start at 0. The following example creates the matrix shown below:

1	2	3
4	5	6

Example of Printing Matrice values:

```
int[][] mat = {{5,7,9,2,1,9},  
               {5,3,4},  
               {3,7,0,8,9}};
```

```
out.println(mat[7/4][0]);  
out.println(mat[1*2][2]);  
out.println(mat.length);  
out.println(mat[0].length);
```



The Output:

```
5  
0  
3  
6
```

You can also set Matrix Values:

```
int[][] mat = new int[3][3];  
mat[0][1]=2;
```

This makes the following matrix:

0	2	0
---	---	---

0	0	0
0	0	0

Traversing 2D Arrays:

Nested Loops are usually used to traverse 2D Arrays. In a row-major traversal, the outer loop iterates through each row, while the inner loop iterates through each column in the row. In a column-major traversal, the order of the loops is reversed.

The number of rows in the 2D array `arr` is `arr.length`. The number columns can be accessed using `arr[x].length`, where `x` is any valid row index.

All the standard 1D array algorithms can be applied to 2D arrays

```
public static double average(int[][] values)
```

```
{ double total = 0;
```

```
int count = 0;
```

```
for (int r = 0; r < values.length; r++)
```

```
{ for (int c = 0; c < values[r].length; c++)
```

```
{ total += values[r][c]; count++; } } }
```

Bonus Matrix algorithm:

```
int[][] mat = { {5,7},{5,3,4,6},{0,8,9}};
```

```
int count = 0;
```

```
for( int r = 0; r < mat.length; r++ )
```

```
{
```

```
    for( int c = 0; c < mat[r].length; c++)
```

```
    {
```

```
        if( mat[r][c] == 5 )
```

```
            count++;
```

```
    }
```

```
}
```

```
System.out.println("5 count = " + count);
```

The Output:

5 count = 2

Example question:

Consider the following code segment, which contains line numbers. The loop is intended to return the sum of both diagonals in a 2d array. A precondition is that the number of rows and columns in the 2d array are equal. However, the program does not produce correct output.

```

/** Preconditions:
 * array.length > 0
 * array.length == array[0].length
 */

public int sumDiagonals(int[][] array)
{ int sum = 0;
  for (int i = 1; i < array.length; i++)
  { for (int j = 1; j < array[0].length; j++)
    { if (i == j || j == array[0].length-1-i)
      { sum += array[i][j]; } }}
  return sum;
}

```

Which of the below solutions will fix the program?

- A. Change the name of the parameter throughout the method to ary.
- B. In lines 4 and 6, initialize the variables i and j to 0 instead of 1.
- C. In lines 4 and 6, change the test of each loop to array[0].length and array.length, respectively.
- D. In line 8, change the first Boolean expression to i != j.
- E. Before the method returns, add a second nested for loop to account for the diagonal that spans from the top-right to the bottom-left corner.

The correct answer is B. Both for loops should be initialized at 0, so that the iteration also includes the first row and first column. Choice A is incorrect because the identifier array is legal in Java. Changing the name would not affect the output of the program. Choice C is incorrect because the change would have no effect on the output of the program. Choice D is incorrect because the current Boolean expression `i == j` is correctly testing whether an element lies on the diagonal that spans from the top-left to the bottom-right. Choice E is incorrect because no second loop is necessary. The disjunction of Boolean expressions on Line 8 tests whether an element lies on either diagonal.

Problem from pg 52: http://support.ebsco.com/LEX/AP-Computer-Science-A_Study-Guide.pdf

Unit 9: Inheritance

Creating Superclasses and Subclasses:

```
class B extends A { }
```

Inheritance essentially copies all of the methods and instance variables from class A and pastes those into class B at *run time*. The code from A is run from within class B.

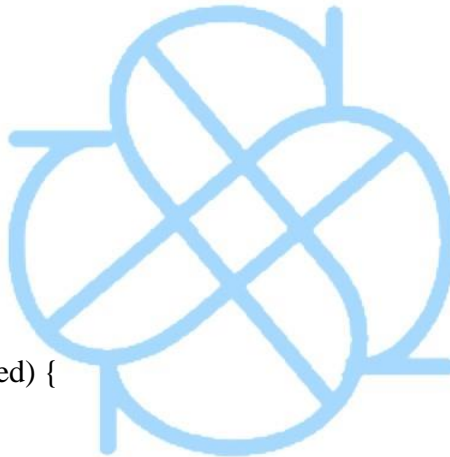
Note: Java does not allow one class to extend more than one other class.

B is said to be a subclass of A. Equivalently, A is a superclass of B. A subclass can only have a single superclass, but a superclass can have many subclasses.

Writing Constructors for Subclasses:

Constructors are not inherited from a superclass. In general, subclass constructors must be explicitly written in the subclass. Within a subclass constructor, the `super` keyword can be used to call a superclass constructor, as in the following example:

```
public class Pet {  
    private String name;  
    public Pet() {  
        name = "";  
    }  
    public Pet(String petName) {  
        name = petName;  
    }  
}  
  
public class Dog extends Pet {  
    public Dog(dogName, dogBreed) {  
        private String breed;  
        super(dogName);  
        breed = dogBreed;  
    }  
}
```



In this example, the `Dog` constructor calls the `Pet` constructor and passes `dogName` to it so that it can be stored in the `name` instance variable. The call to a superclass constructor must be the first line in a subclass constructor. If the superclass has a default (no parameter) constructor, the call to `super` is optional; the default constructor will be called automatically if it is omitted. It is very important to remember that although a subclass inherits all the attributes and behaviors of its superclass, it cannot access private variables or methods from the superclass.

This example and explanation is from <http://support.ebsco.com/LEX/AP-Computer-Science-A-Study-Guide.pdf>

Overriding Methods:

Overriding an inherited method means providing a public method in a subclass with the same method as a public method in the superclass. The method in the subclass will be called instead of the method in the superclass.

When you extend a class, you inherit all methods and instance variables.
You can override the original methods by implementing one with the same signature.

Super Keyword:

super – refers to the parent class
super.toString(); //legal statement
super.super.toString(); //illegal statement

Super is typically used to call parent class constructors and to call parent methods that have been overridden in the sub class.

Super is necessary to distinguish between methods when the subclass and superclass have a method with the same signature.

This example and explanation below is from Westlake Notes:

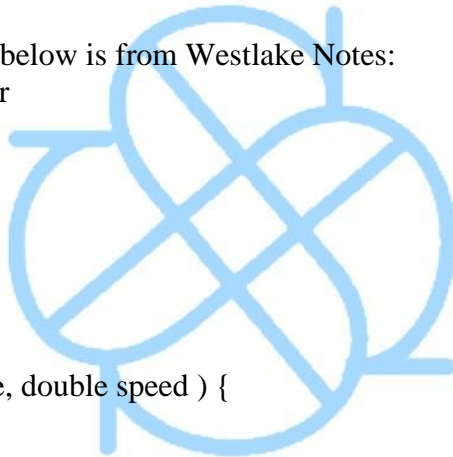
class Skeleton extends Monster

```
{
    private double speed;

    public Skeleton() {
        speed=100;
    }

    public Skeleton( String name, double speed ) {
        super(name);
        this.speed=speed;
    }

    public String toString() {
        return super.toString() + " " + speed;
    }
}
```



In this example above, if a call is not made to a super() constructor, Java will automatically call the default super() constructor on the first line of every subclass constructor.

In the toString() method of Skeleton, the super().toString() is called to retrieve the super class data.

If the super. was not placed in front of the toString() call, a recursive process would begin.

Polymorphism:

This is the ability of one general thing to behave like other specific things.

Ex:

```
public class Mammal{ ... }  
public class Dog extends Mammal{ ... }
```

This creates the following relationship: A Dog is a Mammal.

Because of this relationship, the following assignments are all valid:

```
Mammal test= new Mammal();
```

```
Dog tea= new Dog();
```

```
Mammal rose= new Dog();
```

Remember: a child class can never refer to a parent class but a parent can always refer to a child class.

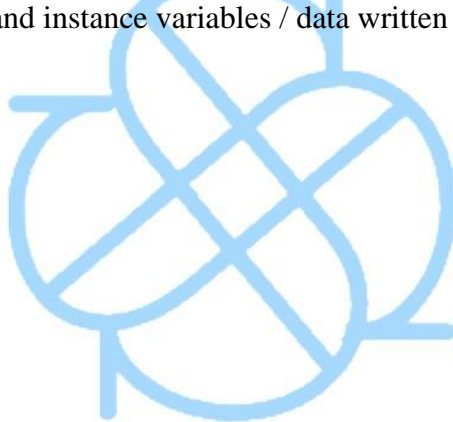
Object Superclass:

Class Object is the one true super class. Object does not extend any other class. All classes extend Object. Class Object is the parent of every other class in Java. All classes in Java start out with all of the same data / instance variables and methods as class Object.

The most common Object methods used are equals(), toString(), clone(), and hashCode(). There are many more as well, but these are the more common ones.

Because String extends Object, String will contain all methods and instance variables / data from Object as well as all methods and instance variables / data written in class String.

String *is an* Object.



Unit 10: Recursion

Recursion:

A recursion method calls itself. A base case in a recursion method is an execution path that does not call itself, all recursive methods must have a base case or else the method will eventually cause an error due to being called too many times.

Every recursive call has its own set of parameters and local variables. When a task is done recursively, the parameters are used to track the progress of the task.

Ex:

```
public class RecursionOne
{
    public void run(int x)
    {
        out.println(x);
        run(x+1);
    }
    public static void main(String args[] )
    {
        RecursionOne test = new RecursionOne();
        test.run(1);
    }
}
```

The Output:

```
1
2
3
4
5
.....
stack overflow
```

This occurs because there is no code or case that makes the recursion stop.

So what does a recursive method need to run correctly?

A recursive method must have a stop condition/ base case.

Recursive calls will continue until the stop condition/ base case is met.

Example with a base case:

```
public class RecursionTwo
```

```

{
    public void run(int x )
    {
        out.println(x);
        if(x<5)          //base case
            run(x+1);
    }
    public static void main(String args[] )
    {
        RecursionTwo test = new RecursionTwo();
        test.run(1);
    }
}

```

The Output:

```

1
2
3
4
5

```

A slightly different example from the previous one:

```

public class RecursionThree
{
    public void run(int x )
    {
        if(x<5)          //base case
            run(x+1);
        out.println(x);
    }
    public static void main(String args[] )
    {
        RecursionThree test = new RecursionThree ();
        test.run(1);
    }
}

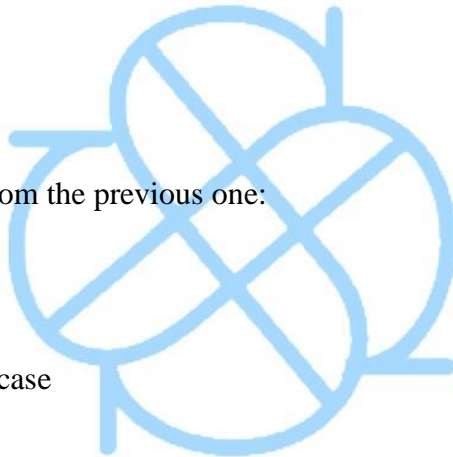
```

The Output:

```

5
4

```



3
2
1

Can also use a do while loop... Example below:

class DoWhile

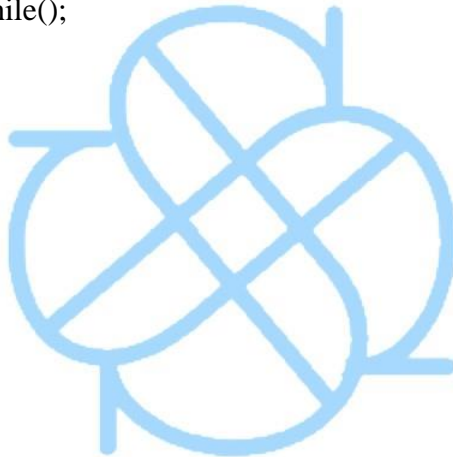
```
{
    public void run( )
    {
        int x=0;
        do{
            x++;
            out.println(x);
        }while(x<10);    //condition
    }
    public static void main(String args[] )
    {
        DoWhile test = new DoWhile();
        test.run( );
    }
}
```

Example of tracing recursion:

int fun(int x, int y)

```
{
    if ( x == 0 )
        return x;
    else
        return x+fun(y-1,x);
}
```

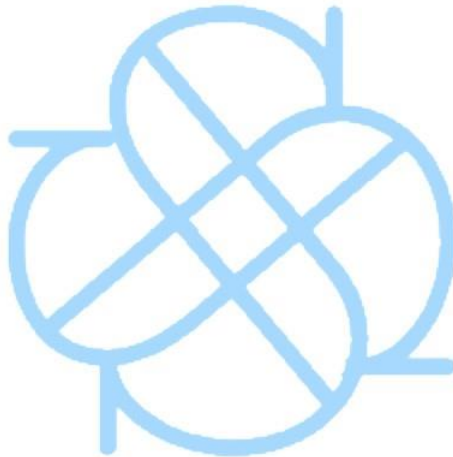
The call (4, 4) will return 16!



Recursive Searching and Sorting:

If an array or ArrayList is sorted, it can be searched using the binary search algorithm, which is more efficient than a linear search and works by keeping track of the beginning and ending positions of the section of the list that remains to be searched.

By taking the average of these two positions, and then examining the item at the position calculated, half of the list can be eliminated in the next iteration. This continues until the item has been found, or until there is no part of the list remaining, in which case the conclusion is that the item does not exist in the list.



“Binary Search:

Step 1: Create three variables, low (the smallest index of the list left to search), high (the largest index of the list left to search), and middle (the middle index of the list with the decimal dropped if necessary)

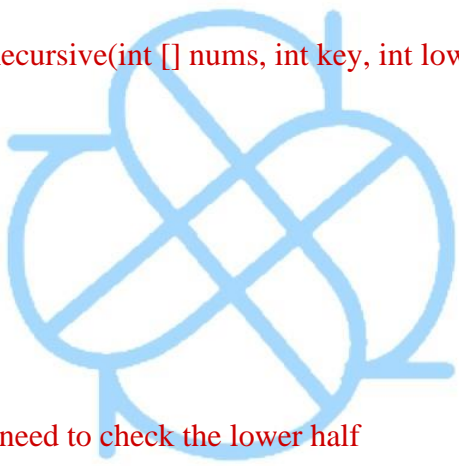
Step 2: Do the following until low is greater than high:

If the middle item is the number, report middle. If the number is less than the middle, the value should be in the lower half of the remaining list so set high to middle minus one. Otherwise, the value should be in the upper half of the remaining list so set low to middle plus one. Then reset the middle to (low plus high) divided by two, the middle of the remaining half of the list.
Step 3: If low becomes greater than high, then the number is not in the list, so report zero.

Binary search is a logarithmic-time algorithm because every time the length of the list doubles, it adds one to the number of values that need to be checked in binary search. Binary Search only works for sorted lists.

Binary Search in Java using Recursion:

```
//Driver Method for binary search recursive function
public static int binarySearchRecursive(int [] nums, int key)
{
    return binarySearchRecursive(nums, key, 0, nums.length - 1);
}
public static int binarySearchRecursive(int [] nums, int key, int low, int high)
{
    if(low > high)
        return -1;
    else
    {
        int mid = (high + low)/2;
        if(nums[mid] == key)
            return mid;
        else if(key < nums[mid]) //need to check the lower half
            return binarySearchRecursive(nums, key, low, mid - 1);
        else //need to check upper half
            return binarySearchRecursive(nums, key, mid + 1, high);
    }
}
```



Sort Algorithms Selection Sort:

- Step 1: Starting with the first thing in the list, find the minimum value in the list.
- Step 2: Swap the first value and the minimum value in the list
- Step 3: Move to the second item in the list and find the minimum value in the remaining unsorted list, then swap it with the second value.
- Step 4: Continue this process until there are no more values in the list to sort.

Insertion Sort:

Step 1: Start with the second item in the list and compare it to the first. If the item is smaller, swap them.

Step 2: Move to the next item in the list. Compare it to the previous item. If the current one is smaller, swap them.

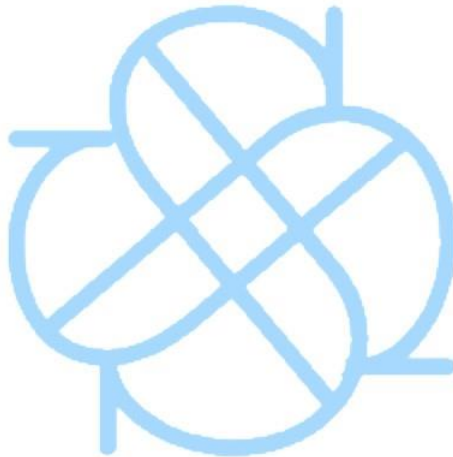
Step 3: Continue swapping until the current item is greater than the previous or there are no more items in the list.

Step 4: Move to the next item and continue the previous process.

Step 5: Do this until all items in the list have been sorted.”[Wood]

EVERYTHING IN THIS COLOR IS FROM:

<http://www.mathwithmrwood.com/ap-computer-science/ap-computer-science-recursion>



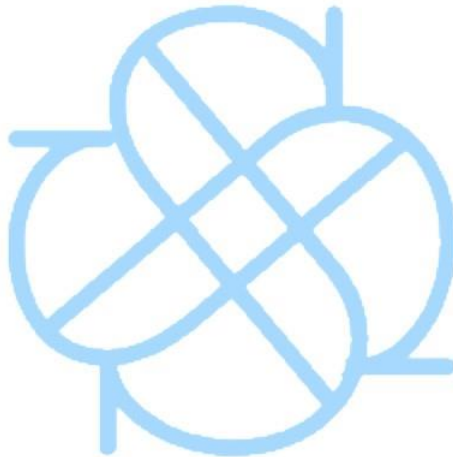
Sample Question:

```
public static int mystery(int n)
{
    if (n <= 1)
    {
        return n;
    }
    else {
        return mystery(n/2) + mystery(n/3);
    }
}
```

Question:

What value is returned as a result of the call `mystery(10)`? A. 8 B. 13 C. 7 D. 4 E. 5

The correct answer is D. The call `mystery(10)` returns the expression `mystery(5)+mystery(3)`, which in turn returns the expression `(mystery(2)+mystery(1))+mystery(3)`, which is expanded to `((mystery(1)+mystery(0))+mystery(1))+mystery(3)`, which evaluates to, via the base case of the recursion, $1+0+1+1+1 = 4$.

**Sources**

EBSCO AP Computer Science A Study Guide

Westlake High Presentations/Notes

MathWithMrWood

Runestone Academy