Unit 1: Primitive Data	
`System.out.print` and `System.out.println`	`print` leaves the cursor at the end of the line and `println` moves the cursor to the next line
`int, double, boolean`	defaults: int = 0 int → double double = 0.0 double !→ int boolean = false
`Integer.MAX_VALUE` `Integer.MIN_VALUE`	The maximum and minimum possible integer value a variable can hold
Primitive vs Reference	Primitives (`int`, `double` `boolean`) store values and References (Strings and Objects) store memory locations
`final` variables	final variables create a constant final classes prevent inheritance final methods prevent overriding
`+, -, *, /, %`	operators
`+=, -=, *=, /=, %=, ++, --`	compound operators
Mixed expressions	int + int results in an int int + double results in a double
`ArithmeticException`	Divide by 0
Rounding	Round up: `(int)(x + 0.5)` Round down: `(int)(x - 0.5)`
Casting	Explicit: `(int) 3.5` → casts to an int `(double) 3` → casts to a double Implicit: `1.0 * 3` → casts to a double
Check for an even number	`num % 2 == 0`

Unit 2: Using Objects	
Object vs Class	An object is a specific instance of a class with defined attributes A class is the formal implementation, or blueprint, of the attributes and behaviors of an object
Object Data: Instance Variables	Created as private at the top of the class. These are the properties of the object
Constructors and Object creation	Default constructor: same name with no parameters. Assigns the instance variables default values Parameter constructor: same name with parameters. Assigns the instance variables to the parameters passed
`null`	used to indicate that a reference is not associated with any object
Object Behavior: Methods	Defined by methods Non-static methods are called through objects of the class

`NullPointerException`	when a method is used on a null reference
	`Dog fido = null;`
	`fido.sit();`

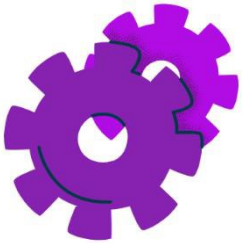
Overloaded	multiple methods with the same name but a different signature
	`public double area(int side)`
	`public double area(int length, int width)`

`\", \\, \n`	Escape sequences used in string literals to print special characters

String in the `java.lang` package available by default	Strings are immutable and can be treated like primitives
	`String first = "this";`
	`String second = first;`
	Any changes made to the variable second will not affect the first variable (not how other references behave)

Know String methods	`length()`
	`substring(int from, int to)`
	`substring(int from)`
	`indexOf(String str)`
	`equals(String other)`
	`compareTo(String other)`

`IndexOutOfBoundsException`	Trying to access an index that is not valid. Indexes go from 0 to `word.length() - 1`
`compareTo(String str)`	Only compare same case letters `"A".compareTo("D")` → negative `"D".compareTo("A")` → positive `"A".compateTo("A")` → zero



Getting a single letter from a word	`substring(index, index + 1)`
Wrapper Class: `Integer` and `Double`	Autoboxing is the automatic conversion that the Java compiler makes between primitive types and their corresponding object wrapper classes. This includes converting an `int` to an `Integer` and a `double` to a `Double`.
Static methods	Methods that are not associated with an object behavior
`Math` Class	`int/double abs(int/double)` `pow(double base, double ex)` `sqrt(double x)` `random()`

`random()`	To get a random integer between `start` and `finish`
	`width = finish - start + 1`
	`(int)(Math.random() * width) + start`

Unit 3: Boolean Expressions and if Statements

==, !=, <, >, <=, >=	Comparison operators between two ints or two doubles
`if`	`if(condition) { … }`
`if-else`	`if(condition) { .. }` `else { … }`

`if-else if`	`if(condition) { … }`
	`else if(condition) { … }`
	`else if(condition) {… }`
	…

Nested ifs	`if(condition) {`
	`  if(condition) { … }`
	`  else { … }`
	`}`
	`else {`
	`  if(condition) { … }`
	`  else { … }`
	`}`

Boolean Operators & Truth Tables	

A	B	A&&B	A || B	!A
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

DeMorgan's Law	"Distribution" of Boolean Operators
	`!(A&&B) = !A || !B`
	`!(A||B) = !A && !B`

== and !=	do not use with references to Strings or Objects; these test aliasing with references

Algorithm: Find if one or more substrings has a particular property	`//count number of "a"s`
	`String word = …;`
	`count = 0;`
	`for(int i = 0; i < word.length(); i++) {`
	`  if(word.substring(i, i + 1).equals("a")) {`
	`  count++;`
	`  }`
	`}`

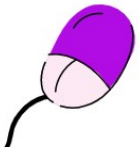
Algorithm: Determine a minimum or maximum value	`min = array[0];`
	`minIndex = 0;`
	`for(int i = 1; i < array.length; i++) {`
	`  if(array[i] < min) {`
	`  min = array[i];`
	`  minIndex = I;`
	`  }`
	`}`

Unit 4: Iteration

`while` loop	`while(condition) { … }` Update the variable in the condition to avoid an infinite loop
`for` loop	`for(initialization; condition; update) { }`
Algorithm: Identify if an integer is or is not evenly divisible by another integer	`if(number % divisor == 0)` `{ /*evenly divisible */ }`
Algorithm: Identify the individual digits in an integer	`number % 10` → ones place `(number / 10) % 10` → tens place

Algorithm: Determine the frequency with which a specific criterion is met	`count = 0`
	`while(condition) {`
	`  if(criterion) {`
	`  count ++`
	`  }`
	`}`

Algorithm: Compute a sum, average, or mode	`sum = 0;`
	`for(int i = 0; i < array.length; i++){`
	`  sum += array[i];`
	`}`



Algorithm: Determine the number of substrings that meet specific criteria	`//count how many "like"s in a sentence`
	`String sentence = …;`
	`for(int i = 0; i < sentence.length() - 4; i++) {`
	`  if(sentence.substring(i, i + 4).equals("like")){`
	`  count++;`
	`  }`
	`}`

Algorithm: Create a new string with the characters reversed	`String original = …;`
	`String reversed = "";`
	`for(int i = original.length(); i > 0; i--){`
	`  reversed += original.substring(i - 1, i);`
	`}`

Unit 5: Writing Classes	
`public` vs `private`	• *public variables* and methods can be accessed anywhere, and by any class. All variables are automatically public. • *private variables* and methods can only be accessed in the class where they are contained.
data encapsulation	A pillar of OOP that protects data from being accessed or modified by any part of a program, except with explicit calls to the accessor and mutator methods.
constructors	• must be public and need to have the same name as the class • default constructor sets all the instance variables to default values • parameter constructors set the instance variables to values that are passed to it

`//`, `/* */`, `/** */`	`//single line comment`
	`/*block of comments that span several lines */`
	`/** Java documentation used on the AP exam */`

Pre and Post Conditions	• *precondition:* condition that must be true just prior to the execution. You do not have to check these in your program. • *postcondition:* condition that must always be true after the execution of a section of program code.

Accessor Methods	`public returnType getVariable()`
	`{ return variable; }`
	
	One for each variable that you will need to access from outside the class.

Mutator Methods	`public void setVariable(variableType name)`
	`{ variable = name; }`
	
	One for each variable that you plan on modifying outside the class.

Static Methods and Variables	Static variables are used as constants in your program:
	`static double PI = 3.1415926;`
	
	Static methods are not associated with an object. They are called on using the class name they are located.
	
	`public static double calcAverage(int a, int b, int c)`

Local vs Global Variables	Local variables can only be used in the block of code that they are declared in. Global variables are declared at the top of the class and can be used at any point in the class.
`this` keyword	Within a non-static method or a constructor, the keyword `this` is a reference to the current object—the object whose method or constructor is being called

Unit 6: Arrays	

Array creation and access	`int[] arr = new int[10];`
	`arr[5] = 9;`
	`int arrayLength = arr.length;`

`ArrayIndexOutOfBounds Exception`	Arrays have indexes from 0 to `arr.length - 1` and trying to access any integer outside of this range will result in this error.
Traversing Arrays	`for(int i = 0; i < arr.length; i ++) { … }` standard loop header to go through the array
Enhanced for loop	`for(int value : arr) { .. }` lets you go through each value in the array arr. An enhanced for loop DOES NOT let you modify the array elements; just look.

Swapping Algorithm	`//swapping two Strings in an array arr`
	`//at indexes 3 and 6`
	`String temp = arr[3];`
	`arr[3] = arr[6];`
	`arr[6] = temp;`

Algorithm: Determine if all elements have a particular property	`//count all the odd integers`
	`count = 0;`
	`for(int i = 0; i < arr.length; i++) {`
	`  if(arr[i] % 2 == 1) {`
	`  count++;`
	`  }`
	`}`

Algorithm: Access all consecutive pairs of elements	`count = 0;`
	`for(int i = 0; i < arr.length - 1; i++) {`
	`  if(arr[i] == arr[i+1]){`
	`  count++;`
	`  }}`

Algorithm: Determine the presence or absence of duplicate elements	`//boolean changes to true if a duplicate is found`
	`boolean found = false;`
	`for(int i = 0; i < arr.length; i++) {`
	`  for(int k = i + 1; k < arr.length; k++) {`
	`  if(arr[i] == arr[k]){`
	`  found = true;`
	`  }`
	`}}`

Algorithm: Shift or rotate elements left or right in an array	`//shifts the elements to the left`
	`int temp = arr[0];`
	`for(int i = 0; i < arr.length - 1; i++) {`
	`  arr[i] = arr[i+1];`
	`}`
	`arr[arr.length - 1] = temp;`

Algorithm: Reverse the order of the elements in an array	`for(int i = 0; i < arr.length/2; i++) {`
	`  int j = arr.length - i - 1;`
	`  int temp = arr[i];`
	`  arr[i] = arr[j];`
	`  arr[j] = temp;`
	`}`

Unit 7: ArrayList

Constructing ArrayLists

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

ArrayList methods

```
myGrades.size()
myGrades.add(obj)
myGrades.add(index, obj)
myGrades.get(index) → returns obj
myGrades.set(index, obj) → returns old obj
myGrades.remove(index) → returns removed obj
```

Traversing ArrayLists

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

Concurrent Modification Exception

```
for(int item: myGrades) {
  myGrades.remove(item);
}
```

Standard algorithms that are used with 1D arrays

Use `list.size()` instead of `array.length`
Use `list.get(i)` instead of `array[i]`
Use `list.set(i, obj)` instead of `array[i] = obj`

Linear Search

```
public static int searchLinear(int num, int[]
list) {
    for(int i = 0; i < list.length; i++){
        if(num == list[i])
             return i;
    }
    return -1;
}
```

Selection Sort

Selects the smallest element to the right of the current index and swaps them.

20	31	6	17	13
6	31	20	17	13
6	13	20	17	31
6	13	17	20	31

Insertion Sort

Current index looks "backwards" through the array to find where it should be inserted at. Elements are shifted from the point of insertion.

20	31	6	17	13
20	31	6	17	13
6	20	31	17	13
6	17	20	31	13
6	13	17	20	31

Unit 8: 2D Arrays

2D arrays are stored as arrays of arrays

```
int[][] matrix = new int[3][5]
```

Row 0					Row 1					Row 2				
C0	C1	C2	C3	C4	C0	C1	C2	C3	C4	C0	C1	C2	C3	C4

Accessing elements of 2d arrays

```
matrix[row index][column index]
```

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]
[4][0]	[4][1]	[4][2]	[4][3]	[4][4]

Traversing Row Major Order

```
for(int r = 0; r < matrix.length; r++) {
   for(int c = 0; c < matrix[0].length; c++) {
      //…
}}
```

Traversing Column Major Order

```
for(int c = 0; c < matrix[0].length; c++) {
   for(int r = 0; r < matrix.length; r++) {
      //…
}}
```

Nested for each loops

```
for(int[] row: matrix) {      //always row major
   for(int element: row) {
      //…
}}
```

Algorithm: Determine if all elements have a particular property

```
//count all the odd integers
count = 0;
for(int i = 0; i < matrix.length; i++){
   for(int k = 0; k < matrix[0].length; k++){
      if(matrix[i][k] % 2 == 1) {
         count++;
      }
}}
```

Algorithm: Printing off the elements of a 2D array

```
for(int i = 0; i < matrix.length; i++){
   for(int k = 0; k < matrix[0].length; k++){
      System.out.print(matrix[i][k]);
   }
   System.out.println();
}
```

Unit 10: Recursion

A recursive method calls itself, and each call has its own local variables. Remember to include a BASE CASE!

```
public static int factorial(int num){
   if(num >= 1)
     return num * factorial(num - 1);
   else
     return 1;
}
```

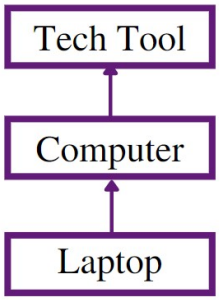
Binary Search

Step 1: Assign the low index to 0 and the high index to the highest index.
Step 2: While the lowest index is less than or equal to the highest index, find the middle index.
Step 3: If you find the value of the number at the middle index, return it.
→ Otherwise, check to see if that value is higher than the middle element.
→ If it is, look through the higher part of the array by making your low index equal to the middle index.
→ Otherwise, look through the lower half of the array by switching the high index to the midpoint.

Merge Sort

- Written recursively
- Don't need to know the code
- Fast on large datasets
- Breaks down an array into single arrays; rebuilds the arrays into a single sorted array

Unit 9: Inheritance




```
public class TechTool
{ /* implementation not shown */ }
public class Computer extends TechTool
{ /* implementation not shown */ }
public class LapTop extends Computer
{ /* implementation not shown */ }
```

- A subclass inherits all instance variables and public methods from their super class.
- Constructors and private methods are NOT inherited.
- To call explicitly on a super's methods or constructors, use the keyword "`super.`" in front of the method name.
- You can only extend one super class.

Cloaking

Legal	Illegal

`TechTool t1 = new Computer();`
`TechTool t2 = new LapTop();`
`Computer c1 = new LapTop();`
	`Computer c2 = new TechTool();`
	`LapTop L1 = new Computer();`
	`LapTop L2 = new TechTool();`

Useful for ArrayLists:

```
ArrayList<TechTool> list = new ArrayList<TechTool>();
list.add(new TechTool());
list.add(new Computer());
list.add(new LapTop());
```

Aliasing

Example	Legal	Illegal

`TechTool a = new TechTool();`
`Computer b = new Computer();`
`LapTop c = new LapTop();`
	`a = b;`
	`b = c;`
	`a = c;`
		`b = a;`
		`c = b;`
		`c = a;`

Polymorphism

- Polymorphism is a process in which a call to an overridden method is resolved at runtime.
- An overridden method is called through the reference variable of a superclass (whatever type the object is DECLARED as, that method needs to be in that class or a parent of that class) at compile time. The program will not compile if this is not satisfied.
- When the program runs, the method that runs is determined from what the object was CREATED as.

```
TechTool t2 = new LapTop();
t2.toString();
```

This will check to make sure that `toString()` is in the `TechTool` class. If it is, it will compile. Then, when the program is executed, it will look in the `LapTop` class first for the `toString()` method and run it if it find it there.

- **object** – An object can have things it knows (attributes) and things it can do (methods). An object is created by a class.
- **class** – A class defines what all objects of that class know (attributes) and can do (methods).
- **parent class (super class)** – the class that another class is inheriting from
- **child class (subclass)** – the class that is doing the inheriting. It inherits access to the object instance variables and methods in the parent class. Private instance variables still have to be called with accessor. Private methods are not inherited.

- **overridden method** – A child class can have the same method signature (method name and parameter list) as a parent class. Since methods are executed starting with the class that created the object (think: what constructor did I use?), that method will be called instead of the inherited parent method, so the child method overrides the parent method.

- **overloaded method** – At least two methods with the same name but different parameter lists. The parameter lists can differ by the number of parameters and/or the types.

```
public static double area(int side)
public static double area(int length, int width)
```

Object Superclass: All classes in Java extend the Object class.

```
//returns a String describing the object
//default implementation: ClassName@MemoryAddress
public String toString()
```

```
//returns true if this object is equal to another
//default implementation only checks for reference equality
public boolean equals(Object other)
```

```
public class Point {
   private int x, y;
   /*constructors and methods not shown */
   public String toString() {
      return x + ", " + y;
   }
}
```

```
public boolean equals(Point other) {
   return this.x == other.x && this.y == other.y;
}
```