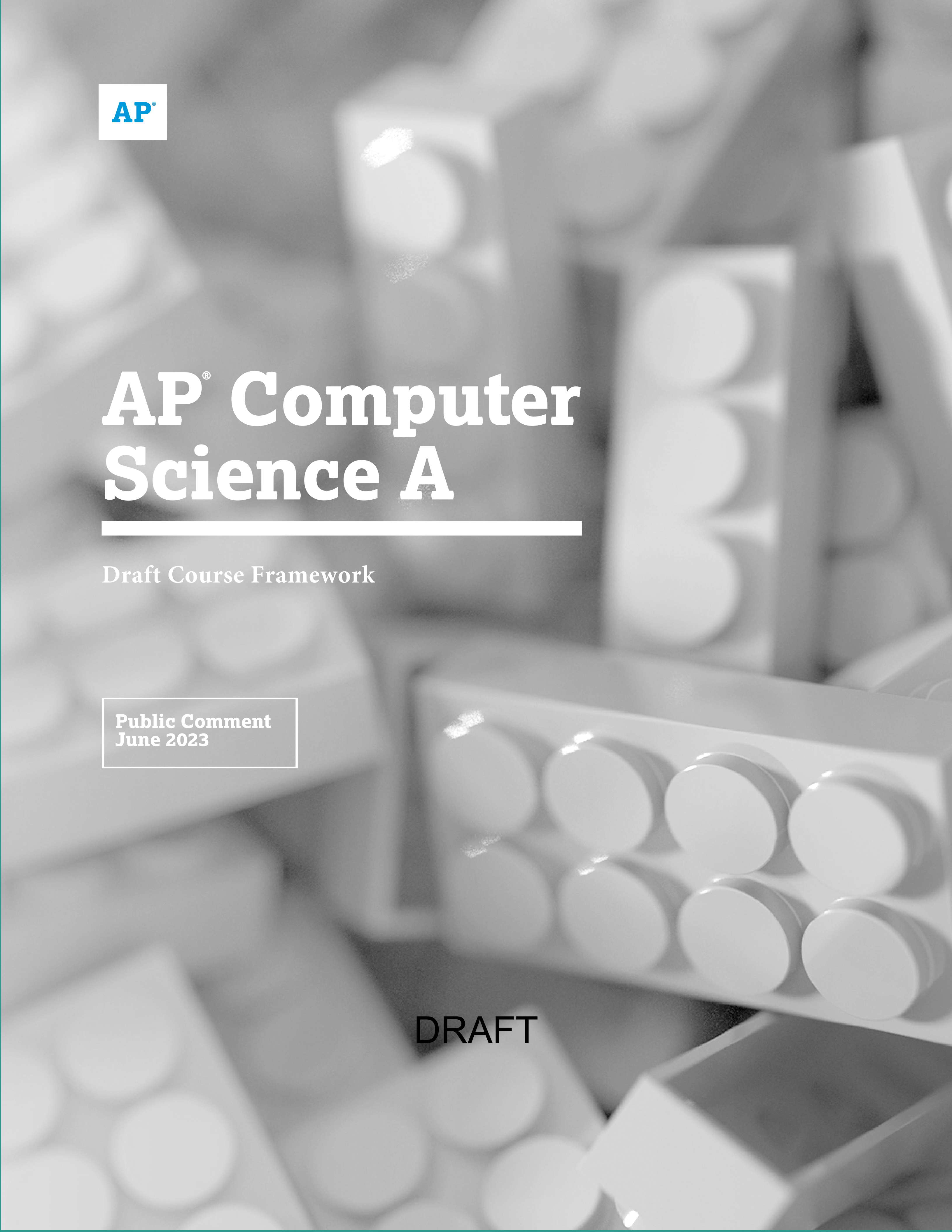

AP® Computer Science A

Draft Course Framework

Public Comment
June 2023

AP Computer Science A
Curriculum Framework

Verb Definitions
Develop code—Design and write/implement code that would be used to meet a specification. Determine the result—Analyze code and describe the value or give the value of the outcome of executing this code. Represent—Use appropriate symbols or words to describe a process or solution to a problem. Identify—Provide a name for the specific topic, without elaboration or explanation. Explain—Provide information about how or why a relationship, situation, or outcome occurs, listing detailed steps of the algorithm or using evidence and/or reasoning. Compare—Identify similarities and differences in code and the outputs that are produced. Calculate—Perform mathematical and logical steps to arrive at a final answer. Describe—Provide the relevant features or characteristics of what the code represents or is being used to accomplish.

Unit 1: Using Objects and Methods

Topic # & Title	Learning Objective	Essential Knowledge
1.1 Introduction to Algorithms	**1.1.A** Represent patterns and algorithms found in everyday life using written language or diagrams.	**1.1.A.1** *Algorithms* define step-by-step processes to follow when completing a task or solving a problem. These algorithms can be represented using written language or diagrams. **1.1.A.2** *Sequencing* defines an order for when steps in a process are completed. Steps in a process are completed one at a time.
1.2 Introduction to Programming and Compilers	**1.2.A** Explain the code compilation and execution process.	**1.2.A.1** Code can be written in any text editor; however, an *Integrated Development Environment (IDE)* is often used to write programs because it provides tools for a programmer to write, compile, and run code. **1.2.A.2** A *compiler* checks code for some errors. Errors detectable by the compiler need to be fixed before the program can be run.
	1.2.B Identify types of programming errors.	**1.2.B.1** A *syntax error* is a mistake in the program where the rules of the programming language are not followed. These errors are detected by the compiler. **1.2.B.2** A *logic error* is a mistake in the algorithm or program that causes it to behave incorrectly or unexpectedly. These errors are detected by testing the program with specific data to see if it produces the expected outcome.

		1.2.B.3 A *run-time* error is a mistake in the program that occurs during the execution of a program. Run-time errors typically cause the program to terminate abnormally. **1.2.B.4** An *exception* is a type of run-time error that occurs as a result of an unexpected error that was not detected by the compiler. It interrupts the normal flow of the program’s execution.
1.3 Variables and Data Types	**1.3.A** Identify the most appropriate data type category for a particular specification.	**1.3.A.1** A *data type* is a set of values and a corresponding set of operations on those values. Data types can be categorized as either primitive or reference. **1.3.A.2** The *primitive data types* used in this course define the set of values and corresponding operations on those values for numbers and Boolean values. **1.3.A.3** A *reference type* is used to define objects that are not primitive types.
	1.3.B Develop code to declare variables to store numbers and Boolean values.	**1.3.B.1** The three primitive data types used in this course are `int`, `double`, and `boolean`. An `int` value is an integer. A `double` value is a real number. A `boolean` value is either `true` or `false`. **EXCLUSION STATEMENT** *The other five primitive data types (long, short, byte, float, and char) are outside the scope of the AP Computer Science A course and exam.* **1.3.B.2** A *variable* is a storage location that holds a value, which can change while the program is running. Every variable has a name and an associated data type. A variable of a primitive type holds a primitive value from that type.
1.4 Expressions and Output	**1.4.A** Develop code to generate output and determine the result that would be displayed.	**1.4.A.1** `System.out.print` and `System.out.println` display information on the computer display. `System.out.println` moves the cursor to a new line after the information has been displayed, while `System.out.print` does not.
	1.4.B Develop code to utilize string literals and determine the result of using string literals.	**1.4.B.1** A *literal* is the source code representation of a fixed value. **1.4.B.2** A string literal is a sequence of characters enclosed in double quotes. **1.4.B.3** *Escape sequences* start with a `\` and have a special meaning in Java. Escape sequences used in this course include `\”`, `\\`, and `\n`.
	1.4.C Develop code for arithmetic expressions and determine the result of these expressions.	**1.4.C.1** Arithmetic expressions include expressions of type `int` and `double`. **1.4.C.2** The arithmetic operators consist of addition `+`, subtraction `-`, multiplication `*`, division `/`, and remainder `%`. An arithmetic operation that uses two `int` values will evaluate to an `int` value. An arithmetic operation that uses a `double` value will evaluate to a `double` value.

		EXCLUSION STATEMENT *Expressions that result in special double values (e.g., infinities and NaN) are outside the scope of the AP Computer Science A course and exam.* **1.4.C.3** When dividing operands that are both `int` values, the result is only the integer portion of the quotient. When dividing operands that use at least one `double` value, the result is the quotient. **1.4.C.4** The remainder operator `%` evaluates to the remainder when `a` is divided by `b`. **EXCLUSION STATEMENT** *The use of values less than 0 for a and the use of values less than or equal to 0 for b is outside the scope of the AP Computer Science A course and exam.* **1.4.C.5** Operators can be used to construct compound expressions. At compile time, operands are associated with operators according to operator precedence to determine how they are grouped. Parentheses can be used to modify operator precedence. **1.4.C.6** An attempt to divide an integer by the integer zero will result in an `ArithmeticException`. **EXCLUSION STATEMENT** *The use of dividing by zero when one operand is a* `double` *is outside the scope of the AP Computer Science A course and exam.*
1.5 Assignment Statements and Input	**1.5.A** Develop code for assignment statements with expressions and determine the value that is stored in the variable as a result of these statements.	**1.5.A.1** Every variable must be assigned a value before it can be used in an expression. That value must be from a compatible data type. A variable is initialized the first time it is assigned a value. **1.5.A.2** The assignment operator `=` allows a program to initialize or change the value stored in a variable. The value of the expression on the right is stored in the variable on the left. **EXCLUSION STATEMENT** *The use of assignment operators inside expressions (e.g., a = b = 4; or a[i += 5]) is outside the scope of the AP Computer Science A course and exam.* **1.5.A.3** During execution, an expression is evaluated to produce a single value. The value of an expression has a type based on the evaluation of the expression.
	1.5.B Develop code to read input.	**1.5.B.1** Input can come in a variety of forms, such as tactile, audio, visual, or text. The `Scanner` class is one way to obtain text input from the keyboard.

		EXCLUSION STATEMENT *Any specific form of input from the user is outside the scope of the AP Computer Science A course and exam.*
1.6 Casting and Range of Variables	**1.6.A** Develop code to cast primitive values to different primitive types in arithmetic expressions and determine the value that is produced as a result.	**1.6.A.1** The *casting operators* `(int)` and `(double)` can be used to convert from a `double` value to an `int` value (or vice versa). **1.6.A.2** Casting a `double` value to an `int` value causes the digits to the right of the decimal point to be truncated. **1.6.A.3** Some code causes `int` values to be automatically cast (widened) to `double` values. **1.6.A.4** Values of type `double` can be rounded to the nearest integer by `(int)(x + 0.5)` for non-negative numbers or `(int)(x – 0.5)` for negative numbers.
	1.6.B Describe conditions when an integer expression evaluates to a value out of range.	**1.6.B.1** The constant `Integer.MAX_VALUE` holds the value of the largest possible `int` value. The constant `Integer.MIN_VALUE` holds the value of the smallest possible `int` value. **1.6.B.2** Integer values in Java are represented by values of type `int`, which are stored using a finite amount (4 bytes) of memory. Therefore, an `int` value must be in the range from `Integer.MIN_VALUE` to `Integer.MAX_VALUE` inclusive. **1.6.B.3** If an expression would evaluate to an `int` value outside of the allowed range, an integer overflow occurs. The result is an `int` value in the allowed range, but not necessarily the value expected.
	1.6.C Describe conditions that limit accuracy of expressions.	**1.6.C.1** Computers allot a specified amount of memory to store data based on the data type. If an expression would evaluate to a `double` that is more precise than can be stored in the allotted amount of memory, a round-off error occurs. The result will be rounded to the representable value. To avoid rounding errors that naturally occur, use `int` values. **EXCLUSION STATEMENT** *Other special decimal data types that can be used to avoid rounding errors are outside the scope of the AP Computer Science A course and exam.*
1.7 Compound Assignment Operators	**1.7.A** Develop code for assignment statements with compound assignment operators and determine the value that is stored in the variable as a result.	**1.7.A.1** *Compound assignment operators* `+=, -=, *=, /=, %=` can be used in place of the assignment operator in numeric expressions. A compound assignment operator performs the indicated arithmetic operation between the value on the left and the value on the right and then assigns the result to the variable on the left.

		1.7.A.2 The post-increment operator `++` and post-decrement operator `--` are used to add `1` or subtract `1` from the stored value of a numeric variable. The new value is assigned to the variable. **EXCLUSION STATEMENT** *The use of increment and decrement operators in prefix form (i.e.,* `++x`*) is outside the scope of this course and AP Exam. The use of increment and decrement operators inside other expressions (i.e.,* `arr[x++]`*) is outside the scope of the AP Computer Science A course and exam.*
1.8 Application Program Interface (API) and Libraries	**1.8.A** Identify the attributes and behaviors of a class found in the libraries contained in an API.	**1.8.A.1** *Libraries* are collections of classes. An *Application Programming Interface (API)* specification informs the programmer how to use those classes. Documentation found in API specifications and libraries is essential to understanding the attributes and behaviors of a class defined by the API. A *class* defines a specific reference type. Classes in the APIs and libraries are grouped into packages. **1.8.A.2** *Attributes* refer to the data related to the class and are stored in variables. *Behaviors* refer to what instances of the class can do (or what can be done with it) and are defined by methods.
1.9 Documentation with Comments	**1.9.A** Describe the functionality and use of code through comments.	**1.9.A.1** Comments are written for other programmers to understand the code and its functionality, but are ignored by the compiler and are not executed when the program is run. Three types of comments in Java include `/* */`, which generates a block of comments, `//`, which generates a comment on one line, and `/** */`, which are Javadoc comments and are used to create API documentation. **1.9.A.2** A *precondition* is a condition that must be true just prior to the execution of a method in order for it to behave as expected. There is no expectation that the method will check to ensure preconditions are satisfied. **1.9.A.3** A *postcondition* is a condition that must always be true after the execution of a method. Postconditions describe the outcome of the execution in terms of what is being returned or the current value of the attributes of an object.
1.10 Method Signatures	**1.10.A** Identify the correct method to call based on documentation and method signatures.	**1.10.A.1** *Procedural abstraction* allows a programmer to use a method by knowing what the method does even if they do not know how the method was written. **1.10.A.2** A *parameter* is a variable declared in the header of a method or constructor and can be used inside the body of the method. This allows values or arguments to be passed and used by a method or constructor. A *method signature* for a method without parameters consists of the method name and an empty parameter list. A

		method signature for a method with parameters consists of the method name and the ordered list of parameter types.
	1.10.B Describe how to call methods.	**1.10.B.1** *A void method* does not have a return value and is therefore not called as part of an expression. **1.10.B.2** *A non-void method* returns a value that is the same type as the return type in the header. To use the return value when calling a non-void method, it must be stored in a variable or used as part of an expression. **1.10.B.3** An *argument* is a value that is passed into a method when the method is called. The arguments passed to a method must be compatible in number and order with the types identified in the parameter list of the method signature. When calling methods, arguments are passed using call by value. *Call by value* initializes the parameters with copies of the arguments. **1.10.B.4** Methods are said to be *overloaded* when there are multiple methods with the same name but different signatures. **1.10.B.5** A method call interrupts the sequential execution of statements, causing the program to first execute the statements in the method before continuing. Once the last statement in the method has been executed or a return statement is executed, the flow of control is returned to the point immediately following where the method was called.
1.11 Calling Class Methods	**1.11.A** Develop code to call class methods.	**1.11.A.1** *Class methods* are associated with the class, not instances of the class. Class methods include the keyword `static` in the header before the method name. **1.11.A.2** Class methods are typically called using the dot operator along with the class name. When the method call occurs in the defining class, the use of the class name is optional in the call.
1.12 Math Class	**1.12.A** Develop code to write expressions that incorporate calls to built-in mathematical libraries and determine the value that is produced as a result.	**1.12.A.1** The `Math` class is part of the `java.lang` package. Classes in the `java.lang` package are available by default. **1.12.A.2** The `Math` class contains only class methods. The following class `Math` methods—including what they do and when they are used—are part of the AP Java Quick Reference: • `static int abs(int x)` returns the absolute value of an `int` value • `static double abs(double x)` returns the absolute value of a `double` value • `static double pow(double base, double exponent)` returns the value of the first parameter raised to the power of the second parameter

		• `static double sqrt(double x)` returns the positive square root of a `double` value • `static double random()` returns a `double` value greater than or equal to `0.0` and less than `1.0` **1.12.A.3** The values returned from `Math.random()` can be manipulated using arithmetic and casting operators to produce a random `int` or `double` in a defined range based on specified criteria. For example, a random even `int` between `10` and `50` or a random `double` between `2.0` inclusive and `5.0` exclusive.
1.13 Objects: Instances of Classes	**1.13.A** Explain the relationship between a class and an object.	**1.13.A.1** 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.
	1.13.B Develop code to declare variables to store reference types.	**1.13.B.1** A variable of a reference type holds an object reference, which can be thought of as the memory address of that object.
1.14 Object Creation and Storage (Instantiation)	**1.14.A** Identify, using its signature, the correct constructor being called.	**1.14.A.1** A class contains *constructors* that are called to create objects. They have the same name as the class. **1.14.A.2** A constructor signature consists of the constructor's name and the ordered list of parameter types. The parameter list, in the header of a constructor, lists the types of the values that are passed and their variable names. **1.14.A.3** Constructors are said to be overloaded when there are multiple constructors with different signatures.
	1.14.B Develop code to declare variables of the correct types to hold object references.	**1.14.B.1** The literal `null` is a special value used to indicate that a reference is not associated with any object. A variable of a reference type holds an object reference or, if there is no object, `null`.
	1.14.C Develop code to create an object by calling a constructor.	**1.14.C.1** An object is typically created using the keyword `new` followed by a call to one of the class's constructors. **1.14.C.2** Parameters allow constructors to accept values to establish the initial values of the attributes of the object. **1.14.C.3** A *constructor argument* is a value that is passed into a constructor when the constructor is called. The arguments passed to a constructor must be compatible in order and number with the types identified in the parameter list in the constructor signature. When calling constructors, arguments are passed using call by value. Call by value initializes the parameters with copies of the arguments.

		1.14.C.4 A constructor call interrupts the sequential execution of statements, causing the program to first execute the statements in the constructor before continuing. Once the last statement in the constructor has been executed, the flow of control is returned to the point immediately following where the constructor was called.
1.15 Calling Instance Methods	**1.15.A** Develop code to call methods and determine the result of these calls.	**1.15.A.1** *Instance methods* are called on objects of the class. The dot operator is used along with the object name to call instance methods. **1.15.A.2** A method call on a `null` reference will result in a `NullPointerException`.
1.16 `String` Manipulation	**1.16.A** Develop code to create string objects and determine the result of creating and combining strings.	**1.16.A.1** A `String` object represents a sequence of characters and can be created by using a string literal. **1.16.A.2** The `String` class is part of the `java.lang` package. Classes in the `java.lang` package are available by default. **1.16.A.3** A `String` object is immutable, meaning once a `String` object is created, its attributes cannot be changed. Methods called on a `String` object do not change the content of the `String` object. **1.16.A.4** Two `String` objects can be concatenated together or combined using the `+` or `+=` operator, resulting in a new `String` object. A primitive value can be concatenated with a `String` object. This causes the implicit conversion of the primitive value to a `String` object. **1.16.A.5** A `String` object can be concatenated with any object, which implicitly calls the object's `toString` method. An object's `toString` method returns a string value representing the object. **EXCLUSION STATEMENT** *Overriding the* `toString` *method is outside the scope of the AP Computer Science A course and exam.*
	1.16.B Develop code to call methods on string objects and determine the result of calling these methods.	**1.16.B.1** A `String` object has index values from 0 to one less than the length of the string. Attempting to access indices outside this range will result in an `IndexOutOfBoundsException`. **1.16.B.2** The following `String` methods—including what they do and when they are used—are part of the AP Java Quick Reference: • `int length()` returns the number of characters in a `String` object • `String substring(int from, int to)` returns the substring beginning at index `from` and ending at index `to − 1`

		• `String substring(int from)` returns `substring(from, length())` • `int indexOf(String str)` returns the index of the first occurrence of `str`; returns `-1` if not found • `boolean equals(String other)` returns `true` if `this` corresponds to the same sequence of characters as `other`; returns `false` otherwise • `int compareTo(String other)` 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`. Strings are ordered based upon the alphabet. **1.16.B.3** A string identical to the single element substring at position `index` can be created by calling `substring(index, index + 1)`.

Unit 2: Selection and Iteration

Topic # & Title	Learning Objective	Essential Knowledge
2.1 Algorithms with Selection and Repetition	**2.1.A** Represent patterns and algorithms that involve selection and repetition found in everyday life using written language or diagrams.	**2.1.A.1** Algorithms can contain selection, through decision-making, and repetition, via looping. **2.1.A.2** The building blocks of algorithms include sequencing, selection, and repetition. **2.1.A.3** *Selection* occurs when a choice of how the execution of an algorithm will proceed is based on a true or false decision. **2.1.A.4** *Repetition* is when a process repeats itself until a desired outcome is reached. **2.1.A.5** The order in which sequencing, selection, and repetition are used contributes to the outcome of the algorithm.
2.2 Boolean Expressions	**2.2.A** Develop code to create Boolean expressions with relational operators and determine the result of these expressions.	**2.2.A.1** Values can be compared using the *relational operators* `==` and `!=` to determine whether the values are the same. With primitive types, this compares the actual primitive values. With reference types, this compares the object references. **2.2.A.2** Numeric values can be compared using the relational operators `<`, `>`, `<=`, and `>=` to determine the relationship between the values. **2.2.A.3** An expression involving relational operators evaluates to a Boolean value.
2.3 `if` Statements	**2.3.A** Develop code to represent branching logical processes by using selection statements and determine the result of these processes.	**2.3.A.1** Selection statements change the sequential execution of statements. **2.3.A.2** An `if` statement is a type of selection statement that affects the flow of control by executing different segments of code based on the value of a Boolean expression. **2.3.A.3** A *one-way selection* (`if` statement) is written when there is a segment of code to execute under a certain condition. In this case, the body is executed only when the Boolean expression is `true`. **2.3.A.4** A *two-way selection* (`if-else` statement) is written when there are two segments of code—one to be executed when the Boolean expression is `true`, and another segment for when the Boolean expression is `false`. In this case, the body of the `if` is executed when the Boolean expression is `true`, and the body of the `else` is executed when the Boolean expression is `false`.
2.4 Nested `if` Statements	**2.4.A** Develop code to represent nested branching logical processes and determine the result of these processes.	**2.4.A.1** Nested `if` statements consist of `if` statements within `if` or `if-else` statements.

		2.4.A.2 The Boolean expression of the inner nested `if` statement is evaluated only if the Boolean expression of the outer `if` statement evaluates to `true`. **2.4.A.3** A *multi-way selection* (`if-else-if`) is written when there are a series of expressions with different segments of code for each condition. Multi-way selection is performed such that no more than one segment of code is executed based on the first expression that evaluates to `true`. If no expression evaluates to `true` and there is a trailing `else` statement, then the body of the `else` is executed.
2.5 Compound Boolean Expressions	**2.5.A** Develop code to represent compound Boolean expressions and determine the result of these expressions.	**2.5.A.1** *Logical operators* `!` (not), `&&` (and), and `||` (or) are used with Boolean values. The order of precedence for evaluating logical operators is `!` (not), `&&` (and), then `||` (or). An expression involving logical operators evaluates to a Boolean value. **2.5.A.2** S*hort-circuit evaluation* occurs when the result of a logical operation using `&&` or `||` can be determined by evaluating only the first Boolean operand, the second is not evaluated.
2.6 Comparing Boolean Expressions	**2.6.A** Compare equivalent Boolean expressions.	**2.6.A.1** Two Boolean expressions are *equivalent* if they evaluate to the same value in all cases. Truth tables can be used to prove Boolean expressions are equivalent. **2.6.A.2** *De Morgan's Law* can be applied to Boolean expressions to create equivalent Boolean expressions. Under De Morgan's Law, the Boolean expression `!(a && b)` is equivalent to `!a || !b` and the Boolean expression `!(a || b)` is equivalent to `!a && !b`.
	2.6.B Develop code to compare object references using Boolean expressions and determine the result of these expressions.	**2.6.B.1** Two different variables can hold references to the same object. Object references can be compared, using `==` and `!=`. **2.6.B.2** A object reference can be compared with `null`, using `==` or `!=`, to determine if the reference actually references an object. **2.6.B.3** Often classes define their own `equals` method, which can be used to specify the criteria for equivalency for two objects of the class. The equivalency of two objects is most often determined using attributes from the two objects. **EXCLUSION STATEMENT** *Overriding the* `equals` *method is outside the scope of the AP Computer Science A course and exam.*
2.7 `while` Loops	**2.7.A** Identify when an iterative process is required to achieve a desired result.	**2.7.A.1** *Iteration* is a form of repetition. Iteration statements change the flow of control by repeating a segment of code zero or more times as long as the Boolean expression controlling the loop evaluates to `true`.

		2.7.A.2 An iteration statement can cause infinite repetition when the Boolean expression always evaluates to `true`. **2.7.A.3** An iteration statement that evaluates the condition before the loop body will not execute the loop at all if the Boolean expression initially evaluates to `false`. **2.7.A.4** Executing a `return` statement inside an iteration statement will halt the iteration and exit the method or constructor. **2.7.A.5** *"Off by one" errors* occur when the iteration statement loops one time too many or one time too few.
	2.7.B Develop code to represent iterative processes using `while` loops and determine the result of these processes.	**2.7.B.1** A `while` loop is a type of iterative statement. In `while` loops, the Boolean expression is evaluated before each iteration of the loop body, including the first. When the expression evaluates to `true`, the loop body is executed. This continues until the Boolean expression evaluates to `false`, whereupon the iteration terminates.
2.8 `for` Loops	**2.8.A** Develop code to represent iterative processes using `for` loops and determine the result of these processes.	**2.8.A.1** A `for` loop is a type of iterative statement. There are three parts in a `for` loop header—the initialization, the Boolean expression, and the update. **2.8.A.2** In a `for` loop, the initialization statement is only executed once before the first Boolean expression evaluation. The variable being initialized is referred to as a loop control variable. The Boolean expression is evaluated immediately after the loop control variable is initialized and then followed by each execution of the increment statement until it is `false`. In each iteration, the update is executed after the entire loop body is executed and before the Boolean expression is evaluated again. **2.8.A.3** A `for` loop can be rewritten into an equivalent `while` loop (and vice versa).
2.9 Implementing Selection and Iteration Algorithms	**2.9.A** Develop code for standard and original algorithms (without data structures) and determine the result of these algorithms.	**2.9.A.1** There are standard algorithms to • identify if an integer is or is not evenly divisible by another integer • identify the individual digits in an integer • determine the frequency with which a specific criterion is met • determine a minimum or maximum value • compute a sum or average
2.10 Implementing String Algorithms	**2.10.A** Develop code for standard and original algorithms that involve strings and determine the result of these algorithms.	**2.10.A.1** There are standard string algorithms to • find if one or more substrings have a particular property • determine the number of substrings that meet specific criteria • create a new string with the characters reversed

2.11 Nested Iteration	**2.11.A** Develop code to represent nested iterative processes and determine the result of these processes.	**2.11.A.1** *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 to its next iteration.
2.12 Informal Run-Time Analysis	**2.12.A** Calculate statement execution counts and informal run-time comparison of iterative statements.	**2.12.A.1** A *statement execution count* indicates the number of times a statement is executed by the program. Statement execution counts are often calculated informally through tracing and analysis of the iterative statements.

Unit 3: Class Creation

Topic # & Title	Learning Objective	Essential Knowledge
3.1 Abstraction and Program Design	**3.1.A** Represent the design of a program by creating diagrams that indicate the classes in the program and the data and procedural abstractions found in each class by including all attributes and behaviors.	**3.1.A.1** *Abstraction* is the process of reducing complexity by focusing on the main idea. By hiding details irrelevant to the question at hand and bringing together related and useful details, abstraction reduces complexity and allows one to focus on the idea. **3.1.A.2** *Data abstraction* provides a separation between the abstract properties of a data type and the concrete details of its representation. Data abstraction manages complexity by giving data a name without referencing the specific details of the representation. Data can take the form of a single variable or a collection of data, such as in a class or a set of data. **3.1.A.3** An attribute is a type of data abstraction that is defined in a class outside any method or constructor. An instance variable is an attribute whose value is unique to each instance of the class. A class variable is an attribute shared by all instances of the class. **3.1.A.4** *Procedural abstraction* provides a name for a process and allows a method to be used only knowing what it does, not how it does it. Through *method decomposition*, a programmer breaks down larger behaviors of the class into smaller behaviors by creating methods to represent each individual smaller behavior. A procedural abstraction may extract shared features to generalize functionality instead of duplicating code. This allows for code reuse, which helps manage complexity. **3.1.A.5** Using parameters allows procedures to be generalized, enabling the procedures to be reused with a range of input values or arguments. **3.1.A.6** Using procedural abstraction in a program allows programmers to change the internals of a method (to make it faster, more efficient, use less storage, etc.) without needing to notify users of the method of the change as long as the method signature and what the method does is preserved. **3.1.A.7** Prior to implementing a class, it is helpful to take time to design each class including its attributes and behaviors. This design can be represented using natural language or diagrams.
3.2 Impact of Program Design	**3.2.A** Explain the social and ethical implications of computing systems.	**3.2.A.1** *System reliability* refers to the program being able to perform its tasks as expected under stated conditions without failure. Programmers should make an

		effort to maximize system reliability by testing the program with a variety of conditions. **3.2.A.2** The creation of programs has impacts on society, economies, and culture. These impacts can be both beneficial and harmful. Programs meant to fill a need or solve a problem can have unintended harmful effects beyond their intended use. **3.2.A.3** Legal issues and intellectual property concerns arise when creating programs. Programmers often reuse code written by others and published as open source and free to use. Incorporation of code that is not published as open source requires the programmer to obtain permission and often purchase the code before integrating it into their program.
3.3 Anatomy of a Class	**3.3.A** Develop code to designate access and visibility constraints to classes, data, constructors, and methods.	**3.3.A.1** A *block of code* is any section of code that is enclosed in braces. Some examples of blocks of code are a class, method, or body of a loop or iterative statement. **3.3.A.2** *Data encapsulation* is a technique in which the implementation details of a class are kept hidden from external classes. The keywords `public` and `private` affect the access of classes, data, constructors, and methods. The keyword `private` restricts access to the declaring class, while the keyword `public` allows access from classes outside the declaring class. **3.3.A.3** In this course, classes are always designated `public`. **3.3.A.4** In this course, constructors are always designated `public`. **3.3.A.5** *Instance variables* belong to the object, and each object has its own copy of the variable. **3.3.A.6** Access to attributes should be kept internal to the class in order to accomplish encapsulation. Therefore, it is good programming practice to designate the instance variables for these attributes as `private` unless the class specification states otherwise. **3.3.A.7** Access to behaviors can be internal or external to the class. Therefore, methods can be designated as either `public` or `private`.
3.4 Constructors	**3.4.A** Develop code to declare instance variables for the attributes to be initialized in the body of the constructors of a class.	**3.4.A.1** An object's *state* refers to its attributes and their values at a given time and is defined by instance variables belonging to the object. This defines a "has-a" relationship between the object and its instance variables. **3.4.A.2** A constructor is used to set the initial state of an object, which should include initial values for all instance variables. When a constructor is called, memory is allocated for the object and the associated object reference is returned. Constructor parameters provide data to initialize instance variables.

		3.4.A.3 When a mutable object is a constructor parameter, the instance variable should be initialized with a copy of the referenced object. In this way, the instance variable does not hold a reference to the original object, and methods are prevented from modifying the state of the original object. **3.4.A.4** When no constructor is written, Java provides a no-parameter constructor, and the instance variables are set to default values according to the data type of the attribute. This constructor is called the *default constructor*. **3.4.A.5** The default value for an attribute of type `int` is `0`. The default value of an attribute of type `double` is `0.0`. The default value of an attribute of type `boolean` is `false`. The default value of a reference type is `null`.
3.5 Methods: How to Write Them	**3.5.A** Develop code to define behaviors of an object through methods written in a class using primitive values and determine the result of calling these methods.	**3.5.A.1** A `void` method does not return a value. Its header contains the keyword `void` before the method name. **3.5.A.2** A non-void method returns a single value. Its header includes the return type in place of the keyword `void`. **3.5.A.3** In non-void methods, a return expression compatible with the return type is evaluated, and the value is returned. This is referred to as “return by value.” **3.5.A.4** The `return` keyword is used to return the flow of control to the point where the method or constructor was called. Any code that is sequentially after a return statement will never be executed. **3.5.A.5** An accessor method allows objects of other classes to obtain a copy of the value of instance variables or class variables. **3.5.A.6** A mutator (modifier) method is a method that changes the values of the instance variables or class variables. A mutator method is often a void method. **3.5.A.7** Methods with parameters receive values through those parameters and use those values in accomplishing the method's task. **3.5.A.8** When an argument is a primitive value, the parameter is initialized with a copy of that value. Changes to the parameter have no effect on the corresponding argument.
3.6 Methods: Passing and Returning References of an Object	**3.6.A** Develop code to define behaviors of an object through methods written in a class using object references and determine the result of calling these methods.	**3.6.A.1** When an argument is an object reference, the parameter is initialized with a copy of that reference; it does not create a new independent copy of the object. If the parameter refers to a mutable object, the method or constructor can use this reference to alter the state of the object. It is good programming practice to not modify mutable objects that are passed as parameters unless required in the specification.

		3.6.A.2 When the return expression evaluates to an object reference, the reference is returned, not a reference to a new copy of the object. **3.6.A.3** Methods cannot access the private data and methods of a parameter that holds a reference to an object unless the parameter is the same type as the method's enclosing class.
3.7 Class Variables and Methods	**3.7.A** Develop code to define behaviors of a class through class methods.	**3.7.A.1** Class methods cannot access or change the values of instance variables or call instance methods without being passed an instance of the class via a parameter. **3.7.A.2** Class methods can access or change the values of class variables and can call other class methods.
	3.7.B Develop code to declare the class variables that belong to the class.	**3.7.B.1** Class variables belong to the class, with all objects of a class sharing a single copy of the class variable. Class variables are designated with the `static` keyword before the variable type. **3.7.B.2** Class variables that are designated `public` are accessed outside of the class by using the class name and the dot operator, since they are associated with a class, not objects of a class. **3.7.B.3** When a variable is declared `final`, its value is not modifiable.
3.8 Scope and Access	**3.8.A** Explain where variables can be used in the code.	**3.8.A.1** Local variables are variables declared in the headers or bodies of blocks of code. Local variables can only be accessed in the block in which they are declared. Since constructors and methods are blocks of code, parameters to constructor or method are also considered local variables. These variables may only be used within the constructor or method and cannot be declared to be `public` or `private`. **3.8.A.2** When there is a local variable or parameter with the same name as an instance variable, the variable name will refer to the local variable instead of the instance variable within the body of the constructor or method.
3.9 `this` Keyword	**3.9.A** Develop code for expressions that are self-referencing and determine the result of these expressions.	**3.9.A.1** Within an instance method or a constructor, the keyword `this` acts as a special variable that holds a reference to the current object—the object whose method or constructor is being called. **3.9.A.2** The keyword `this` can be used to pass the current object as an argument in a method call. **3.9.A.3** Class methods do not have a `this` reference.

Unit 4: Data Collections

Topic # & Title	Learning Objective	Essential Knowledge
4.1 Ethical and Social Issues Around Data Collection	**4.1.A** Explain the risks to privacy from collecting and storing personal data on computer systems.	**4.1.A.1** When using a computer, personal privacy is at risk. When developing new programs, programmers should attempt to safeguard personal privacy of the user. One way to keep personal data secure is to ensure that attributes are encapsulated.
	4.1.B Explain the importance of recognizing data quality and potential issues when using a data set.	**4.1.B.1** *Algorithmic bias* describes systemic and repeated errors in a program that create unfair outcomes for a specific group of users. **4.1.B.2** Programmers should be aware of the data set collection method and the potential for bias when using this method before using the data to extrapolate new information or drawing conclusions. **4.1.B.3** Some data sets are incomplete or contain inaccurate data. Using such data in the development or use of a program can cause the program to work incorrectly or inefficiently.
	4.1.C Identify an appropriate data set to use in order to solve a problem or answer a specific question.	**4.1.C.1** Contents of a data set might be related to a specific question or topic and might not be appropriate to give correct answers or extrapolate information for a different question or topic.
4.2 Introduction to Using Data Sets	**4.2.A** Represent patterns and algorithms that involve data sets found in everyday life using written language or diagrams.	**4.2.A.1** *Data sets* are a collection of specific pieces of information or data. **4.2.A.2** Data sets can be manipulated and analyzed to solve a problem or answer a question. When analyzing data sets, values within the set are accessed and utilized one at a time and then processed according to the desired outcome. **4.2.A.3** Data can be represented in a diagram by using a chart or table. This visual can be used to plan the algorithm that will be used to manipulate the data.
4.3 Array Creation and Access	**4.3.A** Develop code used to represent collections of related data using one-dimensional (1D) array objects.	**4.3.A.1** An array stores multiple values of the same type. The values can be either primitive values or object references. **4.3.A.2** The length of an array is established at the time of creation and cannot be changed. The length of an array can be accessed through the `length` attribute. **4.3.A.3** When an array is created using the keyword `new`, all of its elements are initialized to the default values for the element data type. The default value for `int` is `0`; `double` is `0.0`; `boolean` is `false`; and for a reference type is `null`. **4.3.A.4** Initializer lists can be used to create and initialize arrays. For example, `int[] arr = {1,2,3};`

DRAFT

		4.3.A.5 Square brackets `[]` are used to access and modify an element in a 1D array using an index. **4.3.A.6** The valid index values for an array are 0 through one less than the length of the array, inclusive. Using an index value outside of this range will result in an `ArrayIndexOutOfBoundsException`.
4.4 Array Traversals	**4.4.A** Develop code used to traverse the elements in a 1D array and determine the result of these traversals.	**4.4.A.1** *Traversing an array* is when repetition statements are used to access all or an ordered sequence of elements in an array. **4.4.A.2** Traversing an array with an indexed `for` loop or `while` loop requires elements to be accessed using their indices. **4.4.A.3** An enhanced `for` loop header includes a variable, referred to as the enhanced `for` loop variable. For each iteration of the enhanced `for` loop, the enhanced `for` loop variable is assigned a copy of an element without using its index. **4.4.A.4** Assigning a new value to the enhanced `for` loop variable does not change the value stored in the array. **4.4.A.5** When an array stores object references, the attributes can be modified by calling methods on the enhanced `for` loop variable. This does not change the object references stored in the array. **4.4.A.6** Code written using an enhanced `for` loop to traverse elements in an array can be rewritten using an indexed `for` loop or a `while` loop.
4.5 Implementing Array Algorithms	**4.5.A** Develop code for standard and original algorithms for a particular context or specification that involve arrays and determine the result of these algorithms.	**4.5.A.1** There are standard algorithms that utilize array traversals to • determine a minimum or maximum value • compute a sum or average • determine if at least one element has a particular property • determine if all elements have a particular property • determine the number of elements having a particular property • access all consecutive pairs of elements • determine the presence or absence of duplicate elements • shift or rotate elements left or right • reverse the order of the elements

4.6 Using Text Files	**4.6.A** Develop code to read data from a text file.	**4.6.A.1** A *file* is storage for data that persists when the program is not running. The data in a file can be retrieved during program execution. **4.6.A.2** A file can be connected to the program using the `File` and `Scanner` classes. **4.6.A.3** A file can be opened by creating a `File` object, using the name of the file as the argument of the constructor. • `File(String str)` the `File` constructor that accepts a `String` file name to open for reading. **4.6.A.4** When using the `File` class, it is required to validate the file name being used. One way to accomplish this is to add `throws IOException` to the header of the method that uses the file. If the file name is invalid, the program will terminate. **4.6.A.5** The `File` and `IOException` classes are part of the `java.io` package. An `import` statement must be used to make these classes available for use in the program. **4.6.A.6** The following `Scanner` methods and constructor are used to create `Scanner` objects and read from a file: • `Scanner(File f)` the `Scanner` constructor that accepts a `File` for reading • `int nextInt()` returns the next `int` read from the file or input source if available. If the next `int` does not exist, it will result in an `InputMismatchException` • `double nextDouble()` returns the next `double` read from the file or input source • `boolean nextBoolean()` returns the next `Boolean` read from the file or input source • `String nextLine()` returns the next line of text as a `String` read from the file or input source; returns the empty string if called immediately after another `Scanner` method that is reading from the file or input source • `String next()` returns the next `String` read from the file or input source • `boolean hasNext()` returns `true` if there is a next item to read in the file or input source; `false` otherwise • `void close()` closes this scanner

		EXCLUSION STATEMENT *Accepting input from the keyboard is outside the scope of the AP Computer Science A course and exam.* **4.6.A.7** The following additional `String` methods—including what they do and when they are used—are part of the AP Java Quick Reference: • `String[] split(String del)` returns a `String` array where each element is a substring of `this String` which has been split around matches of the given expression `del` **4.6.A.8** A `while` loop can be used to detect if the file still contains elements to read by using the `hasNext` method as the condition of the loop. **4.6.A.9** A file should be closed when the program is finished using it. The `close` method from `Scanner` is called to close the file.
4.7 Wrapper Classes	**4.7.A** Develop code to use `Integer` and `Double` objects from their primitive counterparts and determine the result of using these objects.	**4.7.A.1** The `Integer` class and `Double` class are part of the `java.lang` package. An `Integer` object is immutable, meaning once an `Integer` object is created, its attributes cannot be changed. A `Double` object is immutable, meaning once a `Double` object is created, its attributes cannot be changed. **4.7.A.2** *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`. The Java compiler applies autoboxing when a primitive value is • passed as a parameter to a method that expects an object of the corresponding wrapper class • assigned to a variable of the corresponding wrapper class **4.7.A.3** *Unboxing* is the automatic conversion that the Java compiler makes from the wrapper class to the primitive type. This includes converting an `Integer` to an `int` and a `Double` to a `double`. The Java compiler applies unboxing when a wrapper class object is • passed as a parameter to a method that expects a value of the corresponding primitive type • assigned to a variable of the corresponding primitive type **4.7.A.4** The following class `Integer` method—including what it does and when it is used—are part of the AP Java Quick Reference: • `static Integer parseInt(String s)` returns the `String` argument as a signed `Integer`

		4.7.A.5 The following class `Double` method—including what it does and when it is used—are part of the AP Java Quick Reference: • `static Double parseDouble(String s)`returns the `String` argument as a signed `Double`
4.8 `ArrayList` Methods	**4.8.A** Develop code for collections of related objects using `ArrayList` objects and the result of calling methods on these objects.	**4.8.A.1** An `ArrayList` object is mutable in size and contains object references. **4.8.A.2** The `ArrayList` constructor `ArrayList()` constructs an empty list. **4.8.A.3** Java allows the generic type `ArrayList<E>`, where the type parameter `E` specifies the type of the elements. When `ArrayList<E>` is specified, the types of the reference parameters and return type when using the `ArrayList` methods are type `E`. `ArrayList<E>` is preferred over `ArrayList`. For example, `ArrayList<String> names = new ArrayList<String>();` allows the compiler to find errors that would otherwise be found at run-time. **4.8.A.4** The `ArrayList` class is part of the `java.util` package. An `import` statement can be used to make this class available for use in the program. **4.8.A.5** The following `ArrayList` methods—including what they do and when they are used—are part of the AP Java Quick Reference: • `int size()` returns the number of elements in the list • `boolean add(E obj)` appends `obj` to end of list; returns `true` • `void add(int index, E obj)` inserts `obj` at position `index (0 <= index <= size)`, moving elements at position `index` and higher to the right (adds 1 to their indices) and adds 1 to size • `E get(int index)` returns the element at position `index` in the list • `E set(int index, E obj)` replaces the element at position `index` with `obj`; returns the element formerly at position `index` • `E remove(int index)` 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`) **4.8.A.6** The indices for an `ArrayList` start at `0` and end at the number of elements `- 1`.
4.9 `ArrayList` Traversals	**4.9.A** Develop code used to traverse the elements of an `ArrayList` and determine the results of these traversals.	**4.9.A.1** Traversing an `ArrayList` is when iteration or recursive statements are used to access all or an ordered sequence of the elements in an `ArrayList`.

		4.9.A.2 Deleting elements during a traversal of an `ArrayList` requires the use of special techniques to avoid skipping elements. **4.9.A.3** Attempting to access an index value outside of its range will result in an `IndexOutOfBoundsException`. **4.9.A.4** Changing the size of an `ArrayList` while traversing it using an enhanced `for` loop can result in a `ConcurrentModifcationException`. Therefore, when using an enhanced `for` loop to traverse an `ArrayList`, you should not add or remove elements.
4.10 Implementing `ArrayList` Algorithms	**4.10.A** Develop code for standard and original algorithms for a particular context or specification that involve `ArrayList` objects and determine the result of these algorithms.	**4.10.A.1** There are standard `ArrayList` algorithms that utilize traversals to • determine a minimum or maximum value • compute a sum or average • determine if at least one element has a particular property • determine if all elements have a particular property • determine the number of elements having a particular property • access all consecutive pairs of elements • determine the presence or absence of duplicate elements • shift or rotate elements left or right • reverse the order of the elements • insert elements • delete elements **4.10.A.2** Some algorithms require multiple `String`, array, or `ArrayList` objects to be traversed simultaneously.
4.11 2D Array Creation and Access	**4.11.A** Develop code used to represent collections of related data using two-dimensional (2D) array objects.	**4.11.A.1** A 2D array is stored as an array of arrays. Therefore, the way 2D arrays are created and indexed is similar to 1D array objects. The size of a 2D array is established at the time of creation and cannot be changed. 2D arrays can store either primitive data or object reference data. **EXCLUSION STATEMENT** *Non-rectangular 2D array objects are outside the scope of the AP Computer Science A course and exam.* **4.11.A.2** When a 2D array is created using the keyword `new`, all of its elements are initialized to the default values for the element data type. The default value for `int` is `0`; `double` is `0.0`; `boolean` is `false`; and for a reference type is `null`.

		4.11.A.3 The initializer list used to create and initialize a 2D array consists of initializer lists that represent 1D arrays. For example, `int[][] arr2D = { {1, 2, 3}, {4, 5, 6} };`. **4.11.A.4** The square brackets `[row][col]` are used to access and modify an element in a 2D array. For the purposes of the exam, when accessing the element at `arr[first][second],` the first index is used for rows, the second index is used for columns. **4.11.A.5** A single array that is a row of a 2D array can be accessed using the 2D array name and a single set of square brackets containing the row index. **4.11.A.6** The number of rows contained in a 2D array can be accessed through the `length` attribute. The valid row index values for a 2D array are 0 through one less than the number of rows or the length of the array, inclusive. The number of columns contained in a 2D array can be accessed through the `length` attribute of one of the rows. The valid column index values for a 2D array are 0 through one less than the number of columns or the length of any given row of the array, inclusive. For example, given a 2D array named `values`, the number of rows is `values.length` and the number of columns is `values[0].length`. Using an index value outside of these ranges will result in an `ArrayIndexOutOfBoundsException`.
4.12 2D Array Traversals	**4.12.A** Develop code used to traverse the elements in a 2D array and determine the result of these traversals.	**4.12.A.1** Nested iteration statements are used to traverse and access all or an ordered sequence of elements in a 2D array. Since 2D arrays are stored as arrays of arrays, the way 2D arrays are traversed using `for` loops and enhanced `for` loops is similar to 1D array objects. Nested iteration statements can be written to traverse the 2D array in "row-major order," "column-major order," or a uniquely defined order. "Row-major order" refers to an ordering of 2D array elements where traversal occurs across each row, where as "column-major order" traversal occurs down each column. **4.12.A.2** The outer loop of a nested enhanced `for` loop used to traverse a 2D array traverses the rows. Therefore, the enhanced `for` loop variable must be the type of each row, which is a 1D array. The inner loop traverses a single row. Therefore, the inner enhanced `for` loop variable must be the same type as the elements stored in the 1D array. Assigning a new value to the enhanced `for` loop variable does not change the value stored in the array.

4.13 Implementing 2D Array Algorithms	**4.13.A** Develop code for standard and original algorithms for a particular context or specification that involve 2D arrays and determine the result of these algorithms.	**4.13.A.1** There are standard algorithms that utilize 2D array traversals to • determine a minimum or maximum value of all the elements or for a designated row, column, or other sub-section • compute a sum or average of all the elements or for a designated row, column, or other sub-section • determine if at least one element has a particular property in the entire 2D array or for a designated row, column, or other sub-section • determine if all elements of the 2D array or a designated row, column, or other sub-section have a particular property • determine the number of elements in the 2D array or in a designated row, column, or other sub-section having a particular property • access all consecutive pairs of elements • determine the presence or absence of duplicate elements in the 2D array or in a designated row, column, or other sub-section • shift or rotate elements in a row left or right or in a column up or down • reverse the order of the elements in a row or column
4.14 Searching Algorithms	**4.14.A** Develop code used for linear search algorithms to search for specific information in a collection and determine the results of executing a search.	**4.14.A.1** *Linear search algorithms* are standard algorithms that check each element in order until the desired value is found or all elements in the array or `ArrayList` have been checked. Linear search algorithms can begin the search process from either end of the array or `ArrayList`. **4.14.A.2** When applying linear search algorithms to 2D arrays, each row must be accessed then linear search applied to each row of the 2D array.
4.15 Sorting Algorithms	**4.15.A** Determine the result of executing sorting algorithms to sort the elements of a collection.	**4.15.A.1** Selection sort and insertion sort are iterative sorting algorithms that can be used to sort elements in an array or `ArrayList`. **4.15.A.2** *Selection sort* repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it into its correct (and final) position in the sorted portion of the list. **4.15.A.3** *Insertion sort* inserts an element from the unsorted portion of a list into its correct (but not necessarily final) position in the sorted portion of the list by shifting elements of the sorted portion to make room for the new element.
4.16 Recursion	**4.16.A** Determine the result of calling recursive methods.	**4.16.A.1** A *recursive method* is a method that calls itself. Recursive methods contain at least one base case, which halts the recursion, and at least one recursive call. Recursion is another form of repetition.

		4.16.A.2 Each recursive call has its own set of local variables, including the parameters. Parameter values capture the progress of a recursive process, much like loop control variable values capture the progress of a loop. **4.16.A.3** Any recursive solution can be replicated through the use of an iterative approach and vice versa. **EXCLUSION STATEMENT** *Writing recursive code is outside the scope of the AP Computer Science A course and exam.*
4.17 Recursive Searching and Sorting	**4.17.A** Determine the result of executing recursive algorithms that use strings or collections.	**4.17.A.1** Recursion can be used to traverse `String` objects, arrays, and `ArrayList` objects.
	4.17.B Determine the result of each iteration of a binary search algorithm used to search for information in a collection.	**4.17.B.1** Data must be in sorted order to use the binary search algorithm. *Binary search* starts at the middle of a sorted array or `ArrayList` and eliminates half of the array or `ArrayList` in each recursive call until the desired value is found or all elements have been eliminated. **4.17.B.2** Binary search is typically more efficient than linear search. **EXCLUSION STATEMENT** *Search algorithms other than linear and binary search are outside the scope of the AP Computer Science A course and exam.* **4.17.B.3** The binary search algorithm can be written either iteratively or recursively.
	4.17.C Determine the result of each iteration of the merge sort algorithm when used to sort a collection.	**4.17.C.1** *Merge sort* is a recursive sorting algorithm that can be used to sort elements in an array or `ArrayList`. **EXCLUSION STATEMENT** *Sorting algorithms other than selection, insertion, and merge sort are outside the scope of the AP Computer Science A course and exam*. **4.17.C.2** Merge sort repeatedly divides an array into smaller subarrays until each subarray is one element and then recursively merges the sorted subarrays back together in sorted order to form the final sorted array.