Teaching Tomorrow's Technology.
CompuScholar

Java Programming for AP Computer Science A

Fall 2025+ Course and Exam Description

Course Syllabus and Planner

Course Overview

This AP Computer Science A class uses the **CompuScholar Java Programming [1]** curriculum as the primary resource. It is taught as a one-year (two-semester) sequence and covers all required topics in the "Computer Science A" Course Description published by the College Board.

Students need to have typical computer usage skills before starting this course; other introductory programming courses are not required. All required concepts are taught from the ground up in a fun, step-by-step manner. The course uses a variety of multimedia content such as full-color, interactive text, narrated instructional videos, and guided classroom discussions. Strong emphasis is placed on hands-on programming labs to demonstrate mastery of lesson concepts.

The **CompuScholar Java Programming** curriculum is **endorsed by the College Board** and is **fully aligned with the AP CS A recommended Unit Sequence**. This allows teachers to easily leverage the additional material and practice questions in the AP Classroom.

Textbook Resource

This course uses the following instructional resource:

[1] **Java Programming** online text, by CompuScholar, Inc. 2025, ISBN 978-1-946113-99-3

https://www.compuscholar.com/schools/courses/java/

Official AP CSA Course and Exam Description

Official AP CSA exam requirements can be found in the AP CSA Course and Exam Description (CED) document published by the College Board.

https://apcentral.collegeboard.org/courses/ap-computer-science-a

Course Material

The course material is designed to appeal to a variety of students, from traditional learners who thrive on written text to audio-visual students who enjoy a multi-media format. All content is delivered through an online system that allows students to work seamlessly both in the classroom and at home.

The course consists of the following student-facing elements:

- **Instructional Videos** – optional (not required) but enjoyed by many students as an audio-visual introduction and reinforcement of the lesson topics.
- **Lesson Text** – required reading, contains full topic details and live coding exercises
- **Quizzes and Exams** – multiple-choice and automatically graded by our system
- **Chapter Homework Exercises** – hands-on practice problems
- **Chapter Activities** – hands-on projects, submitted for a grade

Teachers additionally have access to:

- **Teacher's Guides** – for each lesson, with suggested classroom discussion questions
- **Quiz and Exam Answer Keys** – PDFs for quick reference
- **Activity Solution Guides** – fully coded activity solutions for each chapter activity

Programming Environment and Device Requirements

CompuScholar provides an in-browser Java coding environment. This online feature may be used by students to complete all exercises and activities in all required AP chapters. When using the online coding environment:

- **No local software installation is needed to prepare for the AP exam.**
- **The AP material can be completed from any web browser on any device (including Chromebooks and tablets).**

Later, optional chapters contain a mixture of activities. AP teachers may select any of these topics for students after the AP exam. Some optional activities can be done in CompuScholar's online environment, while others are completed using an external IDE.

When needed, CompuScholar recommends a locally installed JDK and Eclipse platform for an external IDE (see Chapters 34 and 35 for instructions). Teachers may also select any other locally installed or online IDE. **Device requirements for an optional, external IDE depend on the IDE selected.**

Project Grading

Each chapter normally contains hands-on homework exercises and graded activities (labs). The exercises in **all required AP chapters are fully auto graded by the CompuScholar system**. Teachers have complete control over the auto-graded results.

Some activities in later, optional chapters are free-form (creative) or completed in an external IDE. The teacher is responsible for grading those creative or external projects.

Course Outline

The course includes the required content organized into the following units based on the AP Course and Exam Description (CED):

SEMESTER 1

- **CED Unit 1: Using Objects and Methods** – Course Chapters 1 - 8
- **CED Unit 2: Selection and Iteration** - Course Chapters 9 – 15

SEMESTER 2

- **CED Unit 3: Class Creation** - Course Chapters 16 - 20
- **CED Unit 4: Data Collections** – Course Chapters 21 – 29

Chapters 1 and 29 can be flexibly scheduled as time permits. The pages in the subsequent "**Course Planner**" section contain detailed pacing guidance.

Course Navigation

Chapter 1 contains computing, ethics, and security topics recommended (but not tested) by AP CS A and required by many state and national computer science curriculum standards. AP teachers may opt to complete Chapter 1 in sequence or return to the topics after the AP CS A exam.

Chapters 2 – 28 should be completed in sequence and cover all required topics on the AP CS A exam, plus certain other highly recommended software skills. These chapters include substantial, hands-on lab work over the 20-hour minimum AP requirement. The unit projects in chapters 15, 20, and 28 are recommended but may be omitted or postponed if desired (no new skills).

Typical classes will finish all required AP content before the exam administration in May. We recommend using the remaining time before the exam to review the College Board's published practice exams and any other external source of practice FRQ and multiple-choice problems.

Chapter 29 contains a walk-through of the AP CSA exam format, including multiple choice questions (MCQ), free-response questions (FRQ), and AP Classroom resources. This chapter is highly recommended for all AP CSA students and may be introduced by teachers at any time (including earlier in the course).

Chapters 30 – 39 contain optional topics that are not required for AP CS A. Teachers may review and select any of these optional topics for students as time permits after the AP CS A exam. Some optional chapters require the use of an external IDE and/or will be graded by the teacher.

Supplemental Chapters 1 – 4 contain a variety of enrichment topics that may be required by individual states to satisfy requirements for other coding or digital literacy courses. AP teachers may use any of these topics as desired.

Course Planner

The following pages contain an outline of course content by AP CS A unit title and matching textbook chapter. Correlations to the AP CS A **recommended Unit Sequence** are highlighted. Additional, detailed mappings to AP CS A Learning Objectives and Essential Knowledge (LOEKs) are appended at the end of this document.

A typical school year has 36 calendar weeks or 180 days of school. After completing the first 29 chapters, most classes will have several days left for AP exam prep, make-up work, and optional topics. Teachers can use optional material before or after the exam, as time permits.

Each "day" listed below represents one typical day or class period of 45 – 60 minutes. In most cases, students will complete one lesson per day (including the quiz), 1 day per lab, and 1 day per chapter test. Some classes may move faster or slower than the suggested pace.

Semester 1 Timeline

Days	CompuScholar Chapter and Lab	AP CS A Unit Sequence
6	**Chapter 1: Computing Concepts** * Evolution of Computers * Computer Hardware * Computer Software * Computer Ethics * Computer Security	**General curricular requirements (N/A on AP Exam)** Schedule as time permits

Days	CompuScholar Chapter and Lab	AP CS A UNIT 1: Using Objects and Methods
6	**Chapter 2: Getting Started with Java** * Common Programming Languages * The Java Platform * Writing Your First Program * Help and Reference Documentation **LAB: Shopping List**	TOPIC 1.1 - Introduction to Algorithms, Programming, and Compilers
5	**Chapter 3: Data Types and Variables** * Primitive Data Types * Variables * Printing Data **LAB: Treasure Map**	TOPIC 1.1 - Introduction to Algorithms, Programming, and Compilers TOPIC 1.2 - Variables and Data Types

Days	CompuScholar Chapter and Lab	AP CS A UNIT 1: Using Objects and Methods
5	**Chapter 4: Working with Numbers** * Simple Math Operations * Compound Assignments and Shortcuts * Type Casting and Truncation **LAB: Magic Math**	TOPIC 1.1 - Introduction to Algorithms, Programming, and Compilers TOPIC 1.3 - Expressions and Output TOPIC 1.4 - Assignment Statements and Input TOPIC 1.5: Casting and Range of Variables TOPIC 1.6: Compound Assignment Operators
6	**Chapter 5: Introducing Objects** * Java Classes * Reference Variables and Strings * Calling Methods * Documentation with Comments **LAB: Travel Agent**	TOPIC 1.7 - Application Program Interface (API) and Libraries TOPIC 1.8 - Documentation with Comments TOPIC 1.9 - Method Signatures TOPIC 1.12 - Objects: Instances of Classes TOPIC 1.13 - Object Creation and Storage TOPIC 1.14 - Calling Instance Methods
5	**Chapter 6: Using Objects** * Properties and Constructors * Static and Overloaded Methods * User Input with Scanner **LAB: Sketch Robot**	TOPIC 1.4 - Assignment Statements and Input TOPIC 1.7 - Application Program Interface (API) and Libraries TOPIC 1.10 - Calling Class Methods TOPIC 1.13 - Object Creation and Storage
6	**Chapter 7: Working with Strings** * Comparing Strings * Common String Operations * Formatting and Building Strings * Using toString() **LAB: String Theory**	TOPIC 1.15 - String Manipulation

Days	CompuScholar Chapter and Lab	AP CS A UNIT 1: Using Objects and Methods
6	**Chapter 8: Numbering Systems and Java Math** * Java Wrapper Classes & Numeric Conversion * Numbers in Binary, Octal and Hex * Java Math Class * Numeric Limitations **LAB: Math Factory**	TOPIC 1.5: Casting and Range of Variables TOPIC 1.10 - Calling Class Methods TOPIC 4.7 - Wrapper Classes

Days	CompuScholar Chapter and Lab	AP CS A: UNIT 2: Selection and Iteration
6	**Chapter 9: Logic and Decision-Making** * Logical Expressions and Relational Operators * Making Decisions with if() * Using "else-if" and "else" * The "switch" Statement **LAB: Banking System**	TOPIC 2.2 - Boolean Expressions TOPIC 2.3 - if Statements TOPIC 2.4 - Nested if Statements
5	**Chapter 10: More Complex Logic** * Comparing Objects and References * Compound Expressions * Boolean Algebra and Truth Tables **LAB: Wild Card**	TOPIC 2.5 - Compound Boolean Expressions TOPIC 2.6 - Comparing Boolean Expressions
5	**Chapter 11: Handling Exceptions** * Understanding Exceptions * Catching Exceptions * Validating User Input **LAB: Calculator Madness**	**Highly recommended skills as students begin to produce more complex code.**
4	**Chapter 12: Debugging** * Finding Runtime Errors * Debugger Concepts **LAB: Bug Hunt**	**Highly recommended skills as students begin to produce more complex code.**

Days	CompuScholar Chapter and Lab	AP CS A: UNIT 2: Selection and Iteration
6	**Chapter 13: Iteration** * For Loops * While Loops * Continue, Break and Return * Nested Loops **LAB: Fun Factorials**	TOPIC 2.7 - while Loops TOPIC 2.8 - for Loops TOPIC 2.11 - Nested Iteration TOPIC 2.12 - Informal Run-Time Analysis
6	**Chapter 14: Algorithms** * Designing with Flowcharts * Writing Pseudocode * Common Mathematical Algorithms * Common String Algorithms **LAB: Meal Time**	TOPIC 2.1 - Algorithms with Selection and Repetition TOPIC 2.9 - Implementing Selection and Iteration Algorithms TOPIC 2.10 - Implementing String Algorithms
5	**Chapter 15: Algorithm Challenge** * Introducing the Algorithm Challenge **LAB: Challenge Menu** **LAB: Numeric Algorithms** **LAB: String Algorithms**	Schedule as time permits (no new skills)
82	**Total Days, Semester 1**	

Semester 2 Timeline

Days	CompuScholar Chapter and Lab	AP CS A: UNIT 3: Class Creation
7	**Chapter 16: Creating Java Classes** * Object-Oriented Concepts * Defining Classes and Packages * Class Properties * Constructors * Class Methods **LAB: Dog House**	TOPIC 1.9 - Method Signatures TOPIC 3.1 - Abstraction and Program Design TOPIC 3.3 - Anatomy of a Class TOPIC 3.4 – Constructors TOPIC 3.5 - Methods: How to Write Them
7	**Chapter 17: Working with Methods** * Documentation and Design * Variable Scope and Access * Data Encapsulation * Method Overloading * Object Interfaces **LAB: Let's Go Racing!**	TOPIC 3.5 - Methods: How to Write Them TOPIC 3.6 - Methods: Passing and Returning References of an Object TOPIC 3.8 - Scope and Access
5	**Chapter 18: Static Concepts** * Static Properties * Static Methods * Static, Object, and "this" References **LAB: Art School**	TOPIC 3.7 - Class Variables and Methods TOPIC 3.9 - this Keyword
5	**Chapter 19: Program Design Considerations** * System Reliability * Computing Impacts on Society * Coding with AI **LAB: AI Bias Case Study**	TOPIC 3.2 - Impact of Program Design
5	**Chapter 20: Remote Control Project** * Introducing the Remote Control Project **LAB: Creating the Schedule** **LAB: Building a Television** **LAB: Defining the Remote**	Schedule as time permits (no new skills)

Days	CompuScholar Chapter and Lab	AP CS A UNIT 4: Data Collections
7	**Chapter 21: 1D Arrays** * Array Concepts * Array Traversal * Iterators and Enhanced for() loops * Array Algorithms * More Array Algorithms **LAB: Whack-A-Mole**	TOPIC 4.3 - Array Creation and Access TOPIC 4.4 - Array Traversals TOPIC 4.5 - Implementing Array Algorithms
6	**Chapter 22: Lists and ArrayLists** * Java Lists * ArrayLists * Iterators and Enhanced for() Loops * Algorithms with ArrayLists **LAB: Train Yard Jumble**	TOPIC 4.8 - ArrayList Methods TOPIC 4.9 - ArrayList Traversals TOPIC 4.10 - Implementing ArrayList Algorithms
7	**Chapter 23: Searching and Sorting** * Bubble Sort * Selection Sort * Insertion Sort * Sequential and Binary Searches **LAB: Ducks in a Row**	TOPIC 2.12 - Informal Run-Time Analysis TOPIC 4.14 - Searching Algorithms TOPIC 4.15 - Sorting Algorithms
6	**Chapter 24: 2D Arrays** * 2D Arrays * Traversal and Ordering * Array of Arrays * 2D Array Algorithms **LAB: Gold Rush**	TOPIC 4.11 - 2D Array Creation and Access TOPIC 4.12 - 2D Array Traversals TOPIC 4.13 - Implementing 2D Array Algorithms
6	**Chapter 25: File Access** * Data Streams * Reading and Writing Text Data * Reading and Writing Binary Data * Using Scanner to Read Files **LAB: Address CSV**	TOPIC 4.6 - Using Text Files
5	**Chapter 26: Data Management** * Respecting Privacy * Data Quality and Bias * Solving Problems with Data **LAB: Survey Analysis**	TOPIC 4.1 - Ethical and Social Issues Around Data Collection TOPIC 4.2 - Introduction to Using Data Sets

5	**Chapter 27: Recursion** * Recursion * Recursive Binary Search * Merge Sort **LAB: File Explorer**	TOPIC 4.16 – Recursion TOPIC 10.2: Recursive Searching and Sorting
5	**Chapter 28: Image Processing Project** * Image Processing Concepts **LAB: Image Loading** **LAB: Image Histogram** **LAB: Image Compression**	Schedule as time permits (no new skills)

2	**Chapter 29: Preparing for the AP CSA Exam** * Exam Format and Scoring * AP Classroom	Recommended AP Prep – Schedule as desired
78	**Total Days in Semester 2 (all required AP CS A topics complete at this point)**	

Classes who complete the first 28 chapters at this point have spent approximately 160 days and covered all tested AP CS A topics. **The remaining class time should be spent in preparation for the AP exam (including Chapter 29) and any other teacher-selected chapters and lessons.**

Please see the following pages for a description of the optional & supplemental material.

The following table suggests the timeline needed for each **additional or supplemental chapter**, along with notes as to the programming environment and grading approach. There are more chapters available than students can complete in a single year, so teachers can pick topics as time permits!

Days	Optional CompuScholar Material	Notes
5	**Chapter 30: Inheritance** * Superclass and Subclass Concepts * Subclass Constructors * Using Superclass and Subclass References **LAB: Lab Rats**	Auto-graded project
6	**Chapter 31: Polymorphism** * Overriding Superclass Methods * Abstract Classes and Methods * Using Superclass Features with "super" * The "Object" Superclass **LAB: Social Ladder**	Auto-graded project
5	**Chapter 32: Object Composition and Copying** * Functional Decomposition * Composite Classes * Copying Objects **LAB: Designing a Composite Class**	Teacher-graded project
10-15	**Chapter 33: Team Project** * Design Processes and Teamwork * Requirements and Design Documents **LAB: Team Project Requirements** **LAB: Project Design** **LAB: Team Project Implementation** *** Testing Your Code** **LAB: Team Project Testing**	CompuScholar online environment or external IDE, teacher-graded project
3	**Chapter 34: Running Java Locally** * Installing the JDK * Local Source Code * Building and Running from the Command Line	"How-to" chapter to create a local development environment

4	**Chapter 35: The Eclipse IDE** * Introducing Eclipse * Eclipse Java IDE Walk-Through * Creating an Eclipse Project * The Eclipse Debugger	"How-to" chapter to install and use a local IDE
6	**Chapter 36: Graphical Java Programs** * Java Swing * Creating a Simple Window * Event-Driven Programming * Layout Managers **LAB: Phone Dialer**	Requires external IDE (e.g., Eclipse) with Java Swing support. Teacher-graded projects.
5	**Chapter 37: Swing Input Controls** * Text and Numeric Input * List Input * Option Input **LAB: Pizza Place**	Requires external IDE (e.g., Eclipse) with Java Swing support. Teacher-graded projects.
5	**Chapter 38: Vector and Bitmap Images** * Screen Coordinates * Drawing Shapes * Drawing Images **LAB: Sky Art**	Requires external IDE (e.g., Eclipse) with Java Swing support. Teacher-graded projects.
4	**Chapter 39: Program Efficiency** * Algorithm Performance (Big-O) * Measuring Sorting Efficiency **LAB: Comparison of Sorting Algorithms**	External IDE, teacher-graded project
12	**Supplemental Chapter 1: Enrichment Topics** * Encoding Data **Lab: Secret Message** * JavaDoc **Lab: Creating HTML** * Advanced Algorithms * Stock Market Simulation **Lab: Stock Trading** * Stacks and Queues **Lab: Candy Factory** * Exploring UML **Lab: UML Design**	See individual lessons and activities for the programming environment and grading approach.

8	**Supplemental Chapter 2: Software and Industry** * Software Development Process **Lab: Your SDLC Docs** * Software Development Careers **Lab: Career Exploration** * Student Organizations **Lab: CTSO Exploration** * Technical Writing **Lab: Technical Writing**	Offline work, teacher-graded projects
4	**Supplemental Chapter 3: Computers and Modern Society** * Managing your Digital Identity **Lab: Privacy Check-up** * The Impact of Computing **Lab: Impact Analysis** * Artificial Intelligence **Lab: Exploring Self-Driving Cars** * Productivity Tools **Lab: Productivity Report**	Offline work, teacher-graded projects
6	**Supplemental Chapter 4: Computer Networking** * Basic Networking * Network Topology * Network Addressing * Network Design * Network Protocols **Lab: Animal Palace**	Offline work, teacher-graded project

Computational Thinking Practices

The AP Course and Exam Description (CED) lists five "Computational Thinking" practices with itemized skills under each practice. The **Java Programming[1]** course provides hands-on opportunities for students to practice these skills in the form of **in-lesson exercises, homework exercises,** and/or **chapter labs**.

All Computational Thinking skills are reinforced multiple times within the course. The table below describes **one** instance of an in-lesson exercise or chapter lab that supports each skill.

Practice 1: Design Code	**Chapter and Lab**
1.A Determine an appropriate program design to solve a problem or accomplish a task (not assessed).	**Chapter 15 LAB: Algorithm Challenge** – Students will design and implement multiple algorithms to complete specific tasks.
1.B Determine what knowledge can be extracted from data.	**Chapter 26 LAB: Census Analysis** – Students will write analysis logic to extract knowledge from a data set.

Practice 2: Develop Code	**Chapter and Lab**
2.A Write program code to implement an algorithm	**Chapter 14 LAB: Meal Time** – Students will design and implement a program to simulate a specific algorithm.
2.B: Write program code involving data abstractions.	**Chapter 16 LAB: Dog House** – Students will create a new Dog class from scratch, including data attributes.
2.C: Write program code involving procedural abstractions.	**Chapter 17 LAB: Let's Go Racing** - Students will implement multiple methods across 3 classes to satisfy program specifications.

Practice 3: Analyze Code	**Chapter and Lab**
3.A: Determine the result or output based on statement execution order in an algorithm.	**Chapter 10 LAB: Wild Card** – Students will implement a program involving sequential and branching logic, including compound Boolean expressions.
3.B: Determine the result or output based on code that contains data abstractions.	**Chapter 21 LAB: Whack-A-Mole** – Students will implement a program making use of 1D arrays

3.C: Determine the result or output based on code that contains procedural abstractions.	**Chapter 18 LAB: Art School** – Students will complete a program using class and instance methods.
3.D: Explain why a code segment will not compile or work as intended and modify the code to correct the error.	**Chapter 12 LAB: Bug Hunt** – Students will use debugging skills to study an existing program and correct bugs.

Practice 4: Document Code and Computing Systems	**Chapter and Lab**
4.A: Describe the behavior of a code segment or program.	**Chapter 24 Homework: "L4.2: Describe a 2D Array Algorithm"** – Students will study existing code and describe the behavior in their own words.
4.B: Describe the initial conditions that must be met for a code segment to work as intended or described.	**Chapter 24 Homework: "L4.2: Describe a 2D Array Algorithm"** – Students will study existing code and describe the initial conditions that must be met to produce a successful result.

Practice 5: Use Computers Responsibly	**Chapter and Lab**
5.A: Explain how computing impacts society, economy, and culture.	**Chapter 1 and Chapter 26, Homework Exercises** – Students will answer questions about a variety of ethical, cybersecurity, and data analysis scenarios.

The following pages contain detailed cross-reference tables that map every AP Computer Science A topic and essential knowledge to specific course chapters and lessons. For convenience, these cross-references are also available as a separate document at the following online location:

https://www.compuscholar.com/docs/java/AP_Exam_Cross_Reference2025.pdf

CompuScholar, Inc.

Alignment to the College Board AP **Computer Science A**

Learning Objectives and Essential Knowledge (LOEK)

AP Course Details:

Course Title:	AP Computer Science A
Grade Level:	9th - 12th grades
Standards Version:	Fall 2025
Standards Link:	ap-computer-science-a-course-and-exam-description.pdf

CompuScholar Course Details:

Course Title:	Java Programming
Course ISBN:	978-1-946113-99-3
Course Year:	2025

Note 1: Citation(s) listed may represent a subset of the instances where objectives are met throughout the course.

Note 2: Citation(s) for a "Lesson" refer to the "Lesson Text" elements and associated "Activities" within the course, unless otherwise noted. The "Instructional Video" components are supplements designed to introduce or reinforce the main lesson concepts, and the Lesson Text contains full details.

AP Course Description

This course teaches students the fundamentals of the Java programming language and covers all required topics defined by the College Board's **AP Computer Science A** course description.

AP Lab Requirements

The AP Computer Science A course must include a minimum of 20 hours of hands-on structured lab experiences to engage students in individual or group problem-solving.	**This course easily exceeds the 20-hour minimum lab requirement with hands-on lesson exercises and labs in every chapter.**

AP Topics and Essential Knowledge

UNIT 1: Using Objects and Methods	**CITATION(S)**
TOPIC 1.1 - Introduction to Algorithms, Programming, and Compilers	
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.	Chapter 2, Lesson 3

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.	Chapter 2, Lesson 3
1.1.B.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.	Chapter 2, Lesson 2
1.1.B.2 - A *compiler* checks code for some errors. Errors detectable by the compiler need to be fixed before the program can be run.	Chapter 2, Lesson 2
1.1.C.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.	Chapter 2, Lesson 4
1.1.C.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.	Chapter 2, Lesson 4
1.1.C.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.	Chapter 2, Lesson 4
1.1.C.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.	Chapter 2, Lesson 4
TOPIC 1.2 - Variables and Data Types	
1.2.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.	Chapter 3, Lesson 1
1.2.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.	Chapter 3, Lesson 1
1.2.A.3 - A *reference type* is used to define objects that are not primitive types.	Chapter 5, Lesson 2
1.2.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.	Chapter 3, Lesson 1
1.2.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.	Chapter 3, Lessons 1, 2
TOPIC 1.3 - Expressions and Output	
1.3.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.	Chapter 3, Lesson 3
1.3.B.1 - A *literal* is the code representation of a fixed value.	Chapter 2, Lesson 3
1.3.B.2 - A *string literal* is a sequence of characters enclosed in double quotes.	Chapter 2, Lesson 3
1.3.B.3 - *Escape sequences* are special sequences of characters that can be included in a string. They start with a \`\\\` and have a special meaning in Java. Escape sequences used in this course include double quote \`\\"\`, backslash \`\\\\\`, and newline \`\\n\`.	Chapter 3, Lesson 3 Chapter 7, Lesson 4
1.3.C.1 – *Arithmetic expressions*, which consist of numeric values, variables, and operators, include expressions of type \`int\` and \`double\`.	Chapter 4, Lesson 1

1.3.C.2 - The *arithmetic operators* consist of addition `+`, subtraction` -`, multiplication `*`, division `/`, and remainder (modulo) `%`. An arithmetic operation that uses two `int` values will evaluate to an `int` value. An arithmetic operation that uses at least one `double` value will evaluate to a `double` value.	Chapter 4, Lesson 1 Chapter 4, Lesson 3
1.3.C.3 - When dividing numeric values that are both `int` values, the result is only the integer portion of the quotient. When dividing numeric values that use at least one `double` value, the result is the quotient.	Chapter 4, Lesson 1
1.3.C.4 - The remainder (modulo) operator `%` is used to compute the remainder when one number `a` is divided by another number `b`.	Chapter 4, Lesson 1 Chapter 14, Lesson 3
1.3.C.5 - Operators can be used to construct compound expressions. At compile time, numeric values are associated with operators according to operator precedence to determine how they are grouped. Parentheses can be used to modify operator precedence. Multiplication, division, and remainder (modulo) have precedence over addition and subtraction. Operators with the same precedence are evaluated from left to right.	Chapter 4, Lessons 1, 2
1.3.C.6 - An attempt to divide an integer by the integer zero will result in an `ArithmeticException`.	Chapter 4, Lesson 1 Chapter 11, Lesson 1
TOPIC 1.4 - Assignment Statements and Input	
1.4.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. Reference types can be assigned a new object or `null` if there is no object. The literal `null` is a special value used to indicate that a reference is not associated with any object.	Chapter 3, Lesson 2 Chapter 5, Lesson 2
1.4.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.	Chapter 3, Lesson 2 Chapter 4, Lesson 1
1.4.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.	Chapter 4, Lesson 1
1.4.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.	Chapter 6, Lesson 3
TOPIC 1.5: Casting and Range of Variables	
1.5.A.1 - The *casting operators* ` (int) ` and ` (double) ` can be used to convert from a `double` value to an `int` value (or vice versa).	Chapter 4, Lesson 3
1.5.A.2 - Casting a `double` value to an `int` value causes the digits to the right of the decimal point to be truncated.	Chapter 4, Lesson 3
1.5.A.3 - Some code causes `int` values to be automatically cast (widened) to `double` values.	Chapter 4, Lesson 3
1.5.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.	Chapter 4, Lesson 3
1.5.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.	Chapter 8, Lesson 1 Chapter 8, Lesson 4
1.5.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.	Chapter 3, Lesson 1 Chapter 8, Lesson 4

1.5.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.	Chapter 8, Lesson 4
1.5.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.	Chapter 8, Lesson 4
TOPIC 1.6: Compound Assignment Operators	
1.6.A.1 - *Compound assignment operators* `+=`, `-=`, `*=`, `/=`, and `%`= 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.	Chapter 4, Lesson 2
1.6.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.	Chapter 4, Lesson 2
TOPIC 1.7 - Application Program Interface (API) and Libraries	
1.7.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. Existing classes and class libraries can be utilized to create objects.	Chapter 5, Lesson 1 Chapter 6, Lesson 3 Chapter 7, Lesson 2
1.7.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.	Chapter 5, Lessons 1, 3 Chapter 6, Lesson 1
TOPIC 1.8 - Documentation with Comments	
1.8.A.1 - *Comments* are written for both the original programmer and 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.	Chapter 2, Lesson 3 Chapter 5, Lesson 4 Chapter 17, Lesson 1
1.8.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.	Chapter 5, Lesson 4 Chapter 17, Lesson 1
1.8.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.	Chapter 5, Lesson 4 Chapter 17, Lesson 1
TOPIC 1.9 - Method Signatures	
1.9.A.1 - A *method* is a named block of code that only runs when it is called. A *block of code* is any section of code that is enclosed in braces. *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.	Chapter 5, Lesson 3

1.9.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 with parameters consists of the method name and the ordered list of parameter types. A method signature for a method without parameters consists of the method name and an empty parameter list.	Chapter 5, Lesson 3 Chapter 6, Lesson 1
1.9.B.1 - A *void method* does not have a return value and is therefore not called as part of an expression.	Chapter 5, Lesson 3
1.9.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.	Chapter 5, Lesson 3
1.9.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.	Chapter 5, Lesson 3 Chapter 6, Lesson 1
1.9.B.4 - Methods are said to be *overloaded* when there are multiple methods with the same name but different signatures.	Chapter 6, Lesson 2
1.9.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.	Chapter 5, Lesson 3
TOPIC 1.10 - Calling Class Methods	
1.10.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.	Chapter 6, Lesson 2 Chapter 8, Lesson 3
1.10.A.2 - Class methods are typically called using the class name along with the dot operator. When the method call occurs in the defining class, the use of the class name is optional in the call.	Chapter 6, Lesson 2 Chapter 8, Lesson 3
1.11.A.1 - The \`Math\` class is part of the \`java.lang\` package. Classes in the \`java.lang\` package are available by default.	Chapter 8, Lesson 3
TOPIC 1.11 – Math Class	
1.11.A.2 - The \`Math\` class contains only class methods. The following \`Math\` class methods—including what they do and when they are used—are part of the AP Java Quick Reference:	Chapter 8, Lesson 3
* \`static int abs(int x) \` returns the absolute value of an \`int\` value.	Chapter 8, Lesson 3
* \`static double abs(double x)\` returns the absolute value of a \`double\` value.	Chapter 8, Lesson 3
* \`static double pow(double base, double exponent) \` returns the value of the first parameter raised to the power of the second parameter.	Chapter 8, Lesson 3
* \`static double sqrt(double x) \` returns the positive square root of a \`double\` value.	Chapter 8, Lesson 3
* \`static double random()\` returns a \`double\` value greater than or equal to 0.0 and less than 1.0.	Chapter 8, Lesson 3

1.11.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. Each endpoint of the range can be *inclusive*, meaning the value is included, or *exclusive*, meaning the value is not included.	Chapter 8, Lesson 3
TOPIC 1.12 - Objects: Instances of Classes	
1.12.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.	Chapter 5, Lesson 1
1.12.A.2 - A class hierarchy can be developed by putting common attributes and behaviors of related classes into a single class called a *superclass*. Classes that extend a superclass, called *subclasses*, can draw upon the existing attributes and behaviors of the superclass without replacing these in the code. This creates an *inheritance relationship* from the subclasses to the superclass.	Chapter 5, Lesson 1
1.12.A.3 - All classes in Java are subclasses of the `Object` class.	Chapter 5, Lesson 1
1.12.B.1 - A variable of a reference type holds an object reference, which can be thought of as the memory address of that object.	Chapter 5, Lesson 2
TOPIC 1.13 - Object Creation and Storage (Instantiation)	
1.13.A.1 - A class contains *constructors* that are called to create objects. They have the same name as the class.	Chapter 6, Lesson 1
1.13.A.2 - A *constructor signature* consists of the constructor's name, which is the same as the class 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.	Chapter 5, Lesson 3 Chapter 6, Lesson 1
1.13.A.3 - Constructors are said to be overloaded when there are multiple constructors with different signatures.	Chapter 6, Lesson 2
1.13.B.1 - A variable of a reference type holds an object reference or, if there is no object, `null`.	Chapter 5, Lesson 2
1.13.C.1 - An object is typically created using the keyword `new` followed by a call to one of the class's constructors.	Chapter 5, Lesson 2 Chapter 6, Lesson 1
1.13.C.2 - Parameters allow constructors to accept values to establish the initial values of the attributes of the object.	Chapter 6, Lesson 1
1.13.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.	Chapter 6, Lesson 1
1.13.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.	Chapter 6, Lesson 1

TOPIC 1.14 - Calling Instance Methods	
1.14.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.	Chapter 5, Lesson 3
1.14.A.2 - A method call on a `null` reference will result in a `NullPointerException`.	Chapter 5, Lesson 3
TOPIC 1.15 - String Manipulation	
1.15.A.1 - A `String` object represents a sequence of characters and can be created by using a string literal.	Chapter 5, Lesson 2 Chapter 7, Lesson 1
1.15.A.2 - The `String` class is part of the `java.lang` package. Classes in the `java.lang` package are available by default.	Chapter 5, Lessons 1, 2
1.15.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.	Chapter 7, Lesson 2
1.15.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.	Chapter 5, Lesson 2 Chapter 7, Lesson 4
1.15.A.5 - A `String` object can be concatenated with any object, which implicitly calls the object's `toString` method (a behavior which is guaranteed to exist by the inheritance relationship every class has with the `Object` class). An object's `toString` method returns a string value representing the object. Subclasses of `Object` often override the` toString` method with class specific implementation. *Method overriding* occurs when a public method in a subclass has the same method signature as a public method in the superclass, but the behavior of the method is specific to the subclass.	Chapter 7, Lesson 4
1.15.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 a `StringIndexOutOfBoundsException`.	Chapter 7, Lesson 2
1.15.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.	Chapter 7, Lesson 2
* `String substring(int from, int to) ` returns the substring beginning at index from and ending at index `to –1`.	Chapter 7, Lesson 2
* `String substring(int from) ` returns `substring(from,length())`.	Chapter 7, Lesson 2
* `int indexOf(String str) ` returns the index of the first occurrence of `str`; returns `-1` if not found.	Chapter 7, Lesson 2
* `boolean equals(Object other) ` returns `true` if `this` corresponds to the same sequence of characters as other; returns `false` otherwise.	Chapter 7, Lesson 2
* `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.	Chapter 7, Lesson 1
1.15.B.3 - A string identical to the single element substring at position `index` can be created by calling `substring(index, index + 1) `.	Chapter 7, Lesson 2

UNIT 2 - Selection and Iteration	CITATION(S)
TOPIC 2.1 - Algorithms with Selection and Repetition	
2.1.A.1 - The building blocks of algorithms include sequencing, selection, and repetition.	Chapter 14, Lesson 1
2.1.A.2 - Algorithms can contain selection, through decision making, and repetition, via looping.	Chapter 14, Lesson 1
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.	Chapter 14, Lesson 1
2.1.A.4 - *Repetition* is when a process repeats itself until a desired outcome is reached.	Chapter 14, Lesson 1
2.1.A.5 - The order in which sequencing, selection, and repetition are used contributes to the outcome of the algorithm.	Chapter 14, Lesson 1
TOPIC 2.2 - Boolean 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.	Chapter 9, Lesson 1 Chapter 10, Lesson 1
2.2.A.2 - Numeric values can be compared using the relational operators `<`, `>`, `<=`, and `>=` to determine the relationship between the values.	Chapter 9, Lesson 1
2.2.A.3 - An expression involving relational operators evaluates to a Boolean value.	Chapter 9, Lesson 1
TOPIC 2.3 - if Statements	
2.3.A.1 - Selection statements change the sequential execution of statements.	Chapter 9, Lesson 2
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.	Chapter 9, Lesson 2
2.3.A.3 - A *one-way* *selection* (`if` statement) is used 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`.	Chapter 9, Lesson 2
2.3.A.4 - A *two-way* *selection* (`if-else` statement) is used 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`.	Chapter 9, Lesson 3
TOPIC 2.4 - Nested if Statements	
2.4.A.1 - Nested `if` statements consist of `if`, `if-else`, or `if-else-if` statements within `if`, `if-else`, or `if-else-if` statements.	Chapter 9, Lesson 3
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`.	Chapter 9, Lesson 3
2.4.A.3 - A *multiway* selection (`if-else-if`) is used when there are a series of expressions with different segments of code for each condition. Multiway 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.	Chapter 9, Lesson 3

TOPIC 2.5 - Compound Boolean Expressions	
2.5.A.1 - *Logical operators* `!` (not), `&&` (and), and `||` (or) are used with Boolean expressions. The expression `!a` evaluates to `true` if `a` is `false` and evaluates to `false` otherwise. The expression `a && b` evaluates to `true` if both `a` and `b` are `true` and evaluates to `false` otherwise. The expression `a || b` evaluates to `true` if `a` is `true`, `b` is `true`, or both, and evaluates to `false` otherwise. The order of precedence for evaluating logical operators is `!` (not), `&&` (and), then `||` (or). An expression involving logical operators evaluates to a Boolean value.	Chapter 10, Lesson 2
2.5.A.2 - *Short-circuit evaluation* occurs when the result of a logical operation using `&&` or `||` can be determined by evaluating only the first Boolean expression. In this case, the second Boolean expression is not evaluated.	Chapter 10, Lesson 2
TOPIC 2.6 - Comparing 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.	Chapter 10, Lesson 3
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`.	Chapter 10, Lesson 3
2.6.B.1 - Two different variables can hold references to the same object. Object references can be compared using `==` and `!=`.	Chapter 10, Lesson 1
2.6.B.2 - An object reference can be compared with `null`, using `==` or `!=`, to determine if the reference actually references an object.	Chapter 10, Lesson 1
2.6.B.3 - Classes often 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.	Chapter 10, Lesson 1
TOPIC 2.7 - while Loops	
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`.	Chapter 13, Lessons 1, 2
2.7.A.2 - An *infinite loop* occurs when the Boolean expression in an iterative statement always evaluates to `true`.	Chapter 13, Lesson 2
2.7.A.3 - The loop body of an iterative statement will not execute if the Boolean expression initially evaluates to `false`.	Chapter 13, Lesson 2
2.7.A.4 - *Off by one* errors occur when the iteration statement loops one time too many or one time too few.	Chapter 13, Lesson 2
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.	Chapter 13, Lesson 2
TOPIC 4.2: for Loops	
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.	Chapter 13, Lesson 1

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.	Chapter 13, Lesson 1
2.8.A.3 - A `for` loop can be rewritten into an equivalent `while` loop (and vice versa).	Chapter 13, Lesson 2
TOPIC 2.9 - Implementing Selection and Iteration Algorithms	
2.9.A.1 - There are standard algorithms to:	See Below
* identify if an integer is or is not evenly divisible by another integer	Chapter 14, Lesson 3
* identify the individual digits in an integer	Chapter 14, Lesson 3
* determine the frequency with which a specific criterion is met	Chapter 14, Lesson 4
* determine a minimum or maximum value	Chapter 14, Lesson 3
* compute a sum or average	Chapter 13, Lesson 1
TOPIC 2.10 - Implementing String Algorithms	
2.10.A.1 - There are standard string algorithms to:	See Below
* find if one or more substrings have a particular property	Chapter 14, Lesson 4
* determine the number of substrings that meet specific criteria	Chapter 14, Lesson 4
* create a new string with the characters reversed	Chapter 14, Lesson 4
TOPIC 2.11 - Nested Iteration	
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.	Chapter 13, Lesson 4
TOPIC 2.12 - Informal Run-Time Analysis	
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.	Chapter 13, Lesson 1

UNIT 3 - Class Creation	**CITATION(S)**
TOPIC 3.1 - Abstraction and Program Design	
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.	Chapter 16, Lesson 1
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.	Chapter 16, Lesson 1
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.	Chapter 16, Lessons 1, 3 Chapter 18, Lesson 1
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.	Chapter 16, Lessons 1, 5 Chapter 17, Lesson 1
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.	Chapter 16, Lesson 5
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 method users of the change as long as the method signature and what the method does is preserved.	Chapter 16, Lesson 5
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.	Chapter 16, Lesson 1 Chapter 17, Lesson 1
TOPIC 3.2 - Impact of Program Design	
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.	Chapter 19, Lesson 1
3.2.A.2 - The creation of programs has impacts on society, the economy, 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.	Chapter 19, Lesson 2
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.	Chapter 19, Lesson 2

TOPIC 3.3 - Anatomy of a Class	
3.3.A.1 - *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.	Chapter 16, Lesson 1 Chapter 17, Lesson 3
3.3.A.2 - In this course, classes are always designated `public` and are declared with the keyword `class`.	Chapter 16, Lesson 2
3.3.A.3 - In this course, constructors are always designated `public`.	Chapter 16, Lesson 4
3.3.A.4 - *Instance variables* belong to the object, and each object has its own copy of the variable.	Chapter 16, Lesson 3
3.3.A.5 - 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.	Chapter 16, Lessons 1, 3 Chapter 17, Lesson 3
3.3.A.6 - Access to behaviors can be internal or external to the class. Methods designated as `public` can be accessed internally or externally to a class whereas methods designated as `private` can only be accessed internally to the class.	Chapter 16, Lessons 1,5 Chapter 17, Lesson 3
TOPIC 3.4 - Constructors	
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.	Chapter 16, Lesson 3
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, if specified, provide data to initialize instance variables.	Chapter 16, Lesson 4
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.	Chapter 16, Lesson 4
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*.	Chapter 16, Lesson 4
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`.	Chapter 16, Lesson 4
TOPIC 3.5 - Methods: How to Write Them	
3.5.A.1 - A `void` method does not return a value. Its header contains the keyword `void` before the method name.	Chapter 16, Lesson 5
3.5.A.2 - A non-void method returns a single value. Its header includes the return type in place of the keyword `void`.	Chapter 16, Lesson 5
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*.	Chapter 16, Lesson 5 Chapter 17, Lesson 3

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. Executing a return statement inside a selection or iteration statement will halt the statement and exit the method or constructor.	Chapter 13, Lesson 3 Chapter 16, Lesson 5
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. An accessor method is a nonvoid method.	Chapter 16, Lesson 5 Chapter 17, Lesson 3
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.	Chapter 16, Lesson 5 Chapter 17, Lesson 3
3.5.A.7 - Methods with parameters receive values through those parameters and use those values in accomplishing the method's task.	Chapter 16, Lesson 5
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.	Chapter 16, Lesson 5
TOPIC 3.6 - Methods: Passing and Returning References of an Object	
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.	Chapter 16, Lesson 5
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.	Chapter 17, Lesson 3
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.	Chapter 17, Lesson 3
TOPIC 3.7 - Class Variables and 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.	Chapter 18, Lesson 2
3.7.A.2 - Class methods can access or change the values of class variables and can call other class methods.	Chapter 18, Lesson 2
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.	Chapter 18, Lesson 1
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.	Chapter 18, Lesson 1
3.7.B.3 - When a variable is declared `final`, its value cannot be modified.	Chapter 18, Lesson 1

TOPIC 3.8 - Scope and Access	
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 constructors or methods 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`.	Chapter 17, Lesson 2
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.	Chapter 17, Lesson 2
TOPIC 3.9 - this Keyword	
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.	Chapter 18, Lesson 3
3.9.A.2 - The keyword `this` can be used to pass the current object as an argument in a method call.	Chapter 18, Lesson 3
3.9.A.3 - Class methods do not have a `this` reference.	Chapter 18, Lesson 3

UNIT 4 - Data Collections	**CITATION(S)**
TOPIC 4.1 - Ethical and Social Issues Around Data Collection	
4.1.A.1 - When using a computer, personal privacy is at risk. When developing new programs, programmers should attempt to safeguard the personal privacy of the user	Chapter 26, Lesson 1
4.1.B.1 - *Algorithmic bias* describes systemic and repeated errors in a program that create unfair outcomes for a specific group of users.	Chapter 26, Lesson 2
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.	Chapter 26, Lesson 2
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.	Chapter 26, Lesson 2
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.	Chapter 26, Lesson 3
TOPIC 4.2 - Introduction to Using Data Sets	
4.2.A.1 – A *data set* is a collection of specific pieces of information or data.	Chapter 26, Lessons 2, 3
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.	Chapter 26, Lesson 3
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.	Chapter 26, Lesson 3

TOPIC 4.3 - Array Creation and Access	
4.3.A.1 - An *array* stores multiple values of the same type. The values can be either primitive values or object references.	Chapter 21, Lesson 1
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.	Chapter 21, Lesson 1
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`, for `double` is `0.0`, for `boolean` is `false`, and for a reference type is `null`.	Chapter 21, Lesson 1
4.3.A.4 - Initializer lists can be used to create and initialize arrays.	Chapter 21, Lesson 1
4.3.A.5 - Square brackets ` []` are used to access and modify an element in a 1D array using an index.	Chapter 21, Lesson 1
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`.	Chapter 21, Lesson 1
TOPIC 4.4 - Array 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.	Chapter 21, Lesson 2
4.4.A.2 - Traversing an array with an indexed `for` loop or `while` loop requires elements to be accessed using their indices.	Chapter 21, Lesson 2
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.	Chapter 21, Lesson 3
4.4.A.4 - Assigning a new value to the enhanced `for` loop variable does not change the value stored in the array.	Chapter 21, Lesson 3
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.	Chapter 21, Lesson 3
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.	Chapter 21, Lesson 3
TOPIC 4.5 - Implementing Array Algorithms	
4.5.A.1 - There are standard algorithms that utilize array traversals to:	See Below
* determine a minimum or maximum value	Chapter 21, Lesson 4
* compute a sum or average	Chapter 21, Lesson 4
* determine if at least one element has a particular property	Chapter 21, Lesson 4
* determine if all elements have a particular property	Chapter 21, Lesson 4
* determine the number of elements having a particular property	Chapter 21, Lesson 4

* access all consecutive pairs of elements	Chapter 21, Lesson 5
* determine the presence or absence of duplicate elements	Chapter 21, Lesson 5
* shift or rotate elements left or right	Chapter 21, Lesson 5
* reverse the order of the elements	Chapter 21, Lesson 5
TOPIC 4.6 - Using Text Files	
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.	Chapter 25, Lesson 1
4.6.A.2 - A file can be connected to the program using the `File` and `Scanner` classes.	Chapter 25, Lesson 4
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) ` is the `File` constructor that accepts a `String` file name to open for reading, where `str` is the pathname for the file.	Chapter 25, Lesson 4
4.6.A.4 - When using the `File` class, it is required to indicate what to do if the file with the provided name cannot be opened. 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.	Chapter 25, Lesson 4
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.	Chapter 25, Lesson 4
4.6.A.6 - The following `Scanner` methods and constructor—including what they do and when they are used—are part of the AP Java Quick Reference: * `Scanner(File f) ` is 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 or is out of range, it will result in an `InputMismatchException`. * `double nextDouble()` returns the next `double` read from the file or input source. If the next `double` does not exist, it will result in an `InputMismatchException`. * `boolean nextBoolean()` returns the next `boolean` read from the file or input source. If the next `boolean` does not exist, it will result in an `InputMismatchException`. * `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; returns `false` otherwise. * `void close()` closes this scanner.	Chapter 25, Lesson 4
4.6.A.7 - Using `nextLine` and the other `Scanner` methods together on the same input source sometimes requires code to adjust for the methods' different ways of handling whitespace.	Chapter 25, Lesson 4

4.6.A.8 - The following additional `String` method—including what it does and when it is used—is 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`.	Chapter 25, Lesson 4
4.6.A.9 - 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.	Chapter 25, Lesson 4
4.6.A.10 - A file should be closed when the program is finished using it. The `close` method from `Scanner` is called to close the file.	Chapter 25, Lesson 4
TOPIC 4.7 - Wrapper Classes	
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.	Chapter 8, Lesson 1
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	Chapter 8, Lesson 1
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	Chapter 8, Lesson 1
4.7.A.4 - The following class `Integer` method—including what it does and when it is used—is part of the AP Java Quick Reference: * `static int parseInt(String s) ` returns the `String` argument as a signed `int`.	Chapter 8, Lesson 1
4.7.A.5 - The following class `Double` method—including what it does and when it is used—is part of the AP Java Quick Reference: * `static double parseDouble(String s) ` returns the `String` argument as a signed `double`.	Chapter 8, Lesson 1
TOPIC 4.8 - ArrayList Methods	
4.8.A.1 - An `ArrayList` object is mutable in size and contains object references.	Chapter 22, Lesson 2
4.8.A.2 - The `ArrayList` constructor `ArrayList()` constructs an empty list.	Chapter 22, Lesson 2

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.	Chapter 22, Lesson 2
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.	Chapter 22, Lesson 2
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:	See Below
* `int size()` returns the number of elements in the list.	Chapter 22, Lesson 2
* `boolean add(E obj) ` appends `obj` to end of list; returns `true`.	Chapter 22, Lesson 2
* `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.	Chapter 22, Lesson 2
* `E get(int index) ` returns the element at position index in the list.	Chapter 22, Lesson 2
* `E set(int index, E obj) ` replaces the element at position `index` with `obj`; returns the element formerly at position `index`.	Chapter 22, Lesson 2
* `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`.	Chapter 22, Lesson 2
4.8.A.6 - The indices for an `ArrayList` start at `0` and end at the number of elements `- 1`.	Chapter 22, Lesson 2
TOPIC 4.9 - ArrayList 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`.	Chapter 22, Lesson 2
4.9.A.2 - Deleting elements during a traversal of an `ArrayList` requires the use of special techniques to avoid skipping elements.	Chapter 22, Lesson 2
4.9.A.3 - Attempting to access an index value outside of its range will result in an `IndexOutOfBoundsException`.	Chapter 22, Lesson 2
4.9.A.4 - Changing the size of an `ArrayList` while traversing it using an enhanced `for` loop can result in a `ConcurrentModificationException`. Therefore, when using an enhanced `for` loop to traverse an `ArrayList`, you should not add or remove elements.	Chapter 22, Lesson 3

TOPIC 4.10 - Implementing ArrayList Algorithms	
4.10.A.1 - There are standard `ArrayList` algorithms that utilize traversals to:	See Below
* determine a minimum or maximum value	Chapter 22, Lesson 4
* compute a sum or average	Chapter 22, Lesson 4
* determine if at least one element has a particular property	Chapter 22, Lesson 4
* determine if all elements have a particular property	Chapter 22, Lesson 4
* determine the number of elements having a particular property	Chapter 22, Lesson 4
* access all consecutive pairs of elements	Chapter 22, Lesson 4
* determine the presence or absence of duplicate elements	Chapter 22, Lesson 4
* shift or rotate elements left or right	Chapter 22, Lesson 4
* reverse the order of the elements	Chapter 22, Lesson 4
* insert elements	Chapter 22, Lesson 4
* delete elements	Chapter 22, Lesson 4
4.10.A.2 - Some algorithms require multiple `String`, array, or `ArrayList` objects to be traversed simultaneously.	Chapter 22, Lesson 4
TOPIC 4.11 - 2D Array Creation and Access	
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.	Chapter 24, Lesson 3
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`, for `double` is `0.0`, for `boolean` is `false`, and for a reference type is `null`.	Chapter 24, Lesson 1
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} };`.	Chapter 24, Lesson 3
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.	Chapter 24, Lessons 1, 2
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.	Chapter 24, Lesson 2

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`.	Chapter 24, Lessons 1, 2
TOPIC 4.12 - 2D Array 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, whereas *column-major order* traversal occurs down each column.	Chapter 24, Lesson 2
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.	Chapter 24, Lesson 2
OPIC 4.13 - Implementing 2D Array Algorithms	
4.13.A.1 - There are standard algorithms that utilize 2D array traversals to:	See Below
* determine a minimum or maximum value of all the elements or for a designated row, column, or other subsection	Chapter 24, Lesson 4
* compute a sum or average of all the elements or for a designated row, column, or other subsection	Chapter 24, Lesson 4
* determine if at least one element has a particular property in the entire 2D array or for a designated row, column, or other subsection	Chapter 24, Lesson 4
* determine if all elements of the 2D array or a designated row, column, or other subsection have a particular property	Chapter 24, Lesson 4
* determine the number of elements in the 2D array or in a designated row, column, or other subsection having a particular property	Chapter 24, Lesson 4
* access all consecutive pairs of elements	Chapter 24, Lesson 4
* determine the presence or absence of duplicate elements in the 2D array or in a designated row, column, or other subsection	Chapter 24, Lesson 4

* shift or rotate elements in a row left or right or in a column up or down	Chapter 24, Lesson 4
* reverse the order of the elements in a row or column	Chapter 24, Lesson 4
TOPIC 4.14 - Searching Algorithms	
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`.	Chapter 23, Lesson 4
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.	Chapter 24, Lesson 4
TOPIC 4.15 - Sorting Algorithms	
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`.	Chapter 23, Lessons 2-3
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.	Chapter 23, Lesson 2
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.	Chapter 23, Lesson 3
TOPIC 4.16 – Recursion	
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.	Chapter 27, Lesson 1
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.	Chapter 27, Lesson 1
4.16.A.3 - Any recursive solution can be replicated through the use of an iterative approach and vice versa.	Chapter 27, Lesson 1
TOPIC 10.2: Recursive Searching and Sorting	
4.17.A.1 - Recursion can be used to traverse `String` objects, arrays, and `ArrayList` objects.	Chapter 27, Lesson 1
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.	Chapter 23, Lesson 4 Chapter 27, Lesson 2
4.17.B.2 - Binary search is typically more efficient than linear search.	Chapter 23, Lesson 4 Chapter 27, Lesson 2
4.17.B.3 - The binary search algorithm can be written either iteratively or recursively.	Chapter 27, Lesson 2
4.17.C.1 - *Merge sort* is a recursive sorting algorithm that can be used to sort elements in an array or `ArrayList`.	Chapter 27, Lesson 3
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.	Chapter 27, Lesson 3