

First Edition

AP[®] Computer Science A

*Student Edition
Sample Chapter*



**Mc
Graw
Hill**



First Edition

AP[®] Computer Science A

AP Computer Science A

Printed and distributed by McGraw Hill in association with Binary Logic SA.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without prior written permission from the publishers. No part of this work may be used or reproduced in any manner for the purpose of training artificial intelligence technologies or systems.

Disclaimer: McGraw Hill is an independent entity from Microsoft® Corporation and is not affiliated with Microsoft Corporation in any manner. Any Microsoft trademarks referenced herein are owned by Microsoft and are used solely for editorial purposes. This work is in no way authorized, prepared, approved, or endorsed by, or affiliated with, Microsoft.

Please note: This book contains links to websites that are not maintained by the publishers. Although we make every effort to ensure these links are accurate, up-to-date, and appropriate, the publishers cannot take responsibility for the content, persistence, or accuracy of any external or third-party websites referred to in this book, nor do they guarantee that any content on such websites is or will remain accurate or appropriate.

Trademark notice: Product or corporate names mentioned herein may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe. The publishers disclaim any affiliation, sponsorship, or endorsement by the respective trademark owners.

AP®, Advanced Placement®, and College Board® are registered trademarks of the College Board, which was not involved in the production of, and does not endorse, this product.

Cover Credit: © valery_potapova/123rf, maksym2000v/123rf

Copyright © 2026 Binary Logic SA

MHID: 1266511148

ISBN: 9781266511141

Printed in the United States of America.

1 2 3 4 5 6 7 8 9 LHN 30 29 28 27 26 25

mheducation.com **binarylogic.net**



Contents

KEY FEATURES

for AP Success

Course and Exam Overview	24
--------------------------	----

WELCOME TO

AP Computer Science A

What is AP Computer Science A?	34
Course Framework	35
The AP Computer Science A Exam	38
How to Use This Textbook	40
Study Strategies for Success	41
Common Challenges	43
A Journey of Growth	44

FOUNDATIONAL CONCEPTS

in Computer Science

The Essence of Computer Science	46
Computational Thinking and Algorithms	47
Computer Hardware	49
Variables and Data Representation	49
Operating Systems	51
Software Testing and Debugging	55

Boolean Logic in Computer Science	57
Debugging with Trace Tables	62

UNIT 1

Using Objects and Methods 66

Lesson 1

Introduction to Algorithms, Programming, and Compilers 69

Introduction to Programming and Java	69
Algorithms and the Purpose of Programming	70
Why Learn Java?	71
How are Java programs executed?	71
Basic Structure	72
Types of Programming Errors	75
Activities	76
Key Takeaways	77
Key Terms	77

Lesson 2

Variables and Data Types 78

Understanding Variables	78
Primitive Data Types	79
Declaring Variables	80
Variable Naming Rules	81
Using Variables in Expressions	81

Choosing the Right Data Type and Errors to Watch Out For	82
Common errors	82
Activities	84
Key Takeaways	86
Key Terms	86

Lesson 3

Expressions and Output 87

Working with Java's core output methods	87
Evaluating and Debugging Code Structures	88
Expressions in Java	90
Arithmetic Operators	90
Operator Precedence and Associativity	91
String Literals and Escape Sequences	92
Compound Expressions	92
Errors in Expressions	93
Activities	94
Key Takeaways	96
Key Terms	96

Lesson 4

Assignment Statements and Input 97

Assignment Statements	97
Using the Scanner Class for Input	100
Input Validation	102

Activities	106
Key Takeaways	108
Key Terms	108

Lesson 5

Casting and Range of Variables 109

Casting Primitive Values	109
Rounding <code>double</code> Values	110
Integer Overflow and Range of Variables	111
Double Precision and Rounding Errors	112
Activities	113
Key Takeaways	115
Key Terms	115

Lesson 6

Compound Assignment Operators 116

Compound Assignment Operators	116
Increment and Decrement Operators	117
Activities	121
Key Takeaways	123
Key Terms	123

Lesson 7

Application Program Interface (API) and Libraries 124

Understanding APIs	124
Using Java Libraries	125

API Documentation	126
Activities	127
Key Takeaways	128
Key Terms	128

Lesson 8

Documentation with Comments 129

Type of Comment in Java	129
Activities	133
Key Takeaways	134
Key Terms	134

Lesson 9

Method Signatures 135

Methods and Method Signatures	135
Parameters	137
Calling Methods	139
Method Overloading	141
Activities	146
Key Takeaways	149
Key Terms	149

Lesson 10

Calling Class Methods 150

Class Methods	150
Activities	154

Key Takeaways	156
Key Terms	156

Lesson 11

Math Class 157

Calling Math class methods	157
Activities	159
Key Takeaways	160
Key Terms	160

Lesson 12

Objects: Instances and Classes 161

Objects and Classes	161
Reference Variables	165
Activities	166
Key Takeaways	168
Key Terms	168

Lesson 13

Object Creation and Storage (Instantiation) 169

Constructor Signatures	169
Activities	173
Key Takeaways	178
Key Terms	178

Lesson 14

Calling Instance Methods 179

Activities	181
Key Takeaways	183
Key Terms	183

Lesson 15

String Manipulation 184

String fundamentals	184
String Manipulation in Detail	185
Activities	189
Key Takeaways	192
Key Terms	192

Exam Questions 193

UNIT 2

Selection and Iteration 200

Lesson 1

Algorithms with Selection and Repetition 203

Selection and Repetition in Everyday Algorithms	203
The Core Components of Algorithms	204
Activities	207
Key Takeaways	208
Key Terms	208

Lesson 2

Boolean Expressions 209

Understanding Boolean Expressions	209
Relational Operators	209
Comparing Primitive Types vs. Reference Types with Relational Operators	211
Practical Applications of Relational Operators	212
Activities	214
Key Takeaways	215
Key Terms	215

Lesson 3

if Statements 216

Understanding if Statements	216
if-else Statements	217
Using multi-way if-else-if Chains	218
Essential Knowledge for if Statements	220
Activities	222
Key Takeaways	223
Key Terms	223

Lesson 4

Nested if Statements 224

Understanding Nested if Statements	224
Activities	229
Key Takeaways	232

Key Terms	232
Lesson 5	
Compound Boolean Expressions	233
Understanding Logical Operators	233
Examples of Compound Boolean Expressions	234
Short-Circuit Evaluation	236
Activities	240
Key Takeaways	242
Key Terms	242
Lesson 6	
Comparing Boolean Expressions	243
Comparing Boolean Expressions	243
Practical Applications of Boolean Equivalency and De Morgan's Laws	245
Comparing Object References and Determining Equivalency	246
Activities	251
Key Takeaways	253
Key Terms	253
Lesson 7	
while Loops	254
Key Concepts of While Loops	254
Examples of While Loops	255
Activities	258
Key Takeaways	260

Key Terms	260
Lesson 8	
for Loops	262
Structure of a For Loop	262
Examples of For Loops	263
Converting For Loops to While Loops	265
Activities	268
Key Takeaways	270
Key Terms	270
Lesson 9	
Implementing Selection and Iteration Algorithms	271
Standard Algorithms	271
Activities	275
Key Takeaways	277
Lesson 10	
Implementing String Algorithms	278
Essential String Algorithms	278
String Reversal with the substring() method	281
Activities	284
Key Takeaways	287
Key Terms	287
Lesson 11	

Nested Iteration	288
-------------------------	------------

Examples of Nested Iteration	288
------------------------------	-----

Activities	292
------------	-----

Key Takeaways	294
---------------	-----

Key Terms	294
-----------	-----

Lesson 12

Informal Run-Time Analysis	295
-----------------------------------	------------

Statement Execution Counts	295
----------------------------	-----

More Examples on Run-Time Analysis	297
------------------------------------	-----

Activities	299
------------	-----

Key Takeaways	300
---------------	-----

Key Terms	300
-----------	-----

Exam Questions	301
-----------------------	------------

Free Response Question	310
-------------------------------	------------

UNIT 3

Class Creation	314
-----------------------	------------

Lesson 1

Abstraction and Program Design	317
---------------------------------------	------------

What is Abstraction?	317
----------------------	-----

Class Design for Abstraction	321
------------------------------	-----

Procedural Abstraction in Methods	326
-----------------------------------	-----

Examples of Abstraction	327
-------------------------	-----

Activities	329
Key Takeaways	331
Key Terms	331

Lesson 2

Impact of Program Design 332

System Reliability and Testing	332
Social and Ethical Implications	333
Legal Considerations and Intellectual Property	334
Artificial Intelligence in Program Development	334
Activities	337
Key Takeaways	338
Key Terms	339

Lesson 3

Anatomy of a Class 340

Access Modifiers	340
Constructors	344
Encapsulation and Instance Variables	346
Methods and Behavior Access	348
Activities	350
Key Takeaways	353
Key Terms	353

Lesson 4

Constructors 354

Role and Characteristics of Constructors	354
Types of Constructors	356
Handling Mutable Object Parameters in Constructors	358
Constructor Overloading	359
Activities	361
Key Takeaways	364
Key Terms	364

Lesson 5

Methods: How to Write Them 365

Void vs non-void Methods	365
The <code>return</code> Keyword	366
Accessor vs Mutator Methods	367
Parameters in Methods	370
Activities	372
Key Takeaways	375
Key Terms	375

Lesson 6

Methods: Passing and Returning References of an Object 376

Passing Object References as Parameters	376
Returning Object References	378
Method Access Restrictions with Object Parameters	380
Best Practices: Methods - Passing and Returning References	382
Activities	389

Key Takeaways	393
Key Terms	393

Lesson 7

Class Variables and Methods 394

Instance vs Class Variables	394
Class Methods	397
Accessing Class Variables and Methods	398
Final Class Variables	399
Activities	400
Key Takeaways	401
Key Terms	401

Lesson 8

Scope and Access 402

Local Variables	402
Instance Variables	403
Class Variables	404
Activities	405
Key Takeaways	407
Key Terms	407

Lesson 9

this Keyword 408

The Role of <code>this</code> in Instance Methods	408
Using <code>this</code> to Call Another Constructor	409

Passing <code>this</code> as a Parameter	410
<code>this</code> in Class Methods	411
Activities	414
Key Takeaways	417
Key Terms	417
Exam Questions	418
Free Response Question	432
UNIT 4	
Data Collections	434
Lesson 1	
Ethical and Social Issues Around Data Collection	438
Privacy Risks in Data Collection	438
Data Quality and Algorithmic Bias	439
Sources of Algorithmic Bias	439
Mitigating Algorithmic Bias	440
Attributes of High-Quality Data	440
Strategies for Ensuring Data Quality	440
Identifying Relevant Data Sets	441
Activities	442
Key Takeaways	443
Key Terms	443

Lesson 2

Working with Data Sets 444

Understanding Data Sets	444
Organization and Structure	445
Common Operations on Data Sets	445
Challenges in Working with Data Sets	446
Activities	447
Key Takeaways	448
Key Terms	448

Lesson 3

Array Creation and Access 449

What is an Array?	449
Declaring and Initializing Arrays	453
Accessing and Modifying Array Elements	455
Activities	457
Key Takeaways	459
Key Terms	459

Lesson 4

Array Traversals 460

The basics of array traversals	460
Dynamic Initialization	467
Activities	469
Key Takeaways	471

Key Terms	471
Lesson 5	
Implementing Array Algorithms	472
Standard Algorithms	472
More Advanced Algorithms	477
Activities	486
Key Takeaways	489
Key Terms	489
Lesson 6	
Using Text Files	490
Reading Data from a File	490
Scanner methods	493
String <code>split(String del)</code> method	497
Activities	501
Key Takeaways	503
Key Terms	504
Lesson 7	
Wrapper Classes	505
Introduction to Wrapper Classes	505
Autoboxing and Unboxing	506
Common Methods in Wrapper Classes	506
Immutability of Wrapper Classes	507
Activities	508

Key Takeaways	509
Key Terms	509

Lesson 8

ArrayList Methods 510

Introduction to ArrayLists	510
Key ArrayList Methods	511
Dynamic Nature	514
Generics	515
Zero-Based Indexing	516
Activities	517
Key Takeaways	519
Key Terms	519

Lesson 9

ArrayList Traversals 520

Basic ArrayList traversals	520
Advanced ArrayList traversals	523
Using a backward <code>for</code> loop	525
Handling <code>ConcurrentModificationExceptions</code>	526
Activities	527
Key Takeaways	529
Key Terms	529

Lesson 10

Implementing ArrayList Algorithms 530

ArrayList Algorithms	530
Working with ArrayLists of objects	540
Simultaneous traversal of multiple lists	541
Activities	543
Key Takeaways	546
Key Terms	546

Lesson 11

2D Array Creation and Access 547

Defining a 2D Array	547
Accessing and Modifying Elements	548
Examples of Working with 2D Arrays	549
Activities	552
Key Takeaways	554
Key Terms	554

Lesson 12

2D Array Traversals 555

Basics of 2D Array Traversals	555
Activities	562
Key Takeaways	564
Key Terms	564

Lesson 13

Implementing 2D Array Algorithms 565

Array Standard Algorithms	565
---------------------------	-----

Activities	579
Key Takeaways	581
Key Terms	581

Lesson 14

Search Algorithms 582

Linear Search Algorithms	582
Binary Search	585
Activities	590
Key Takeaways	593
Key Terms	594

Lesson 15

Sorting Algorithms 595

Selection Sort	595
Insertion Sort	598
Activities	603
Key Takeaways	605
Key Terms	605

Lesson 16

Recursion 606

How Recursion works	606
Activities	616
Key Takeaways	618
Key Terms	618

Lesson 17

Recursive Searching and Sorting 619

Recursive Search 619

Recursive Merge Sort 623

Activities 629

Key Takeaways 631

Key Terms 631

Exam Questions 632

Free Response Questions 643

PRACTICE EXAM

SECTION I

Multiple Choice 648

SECTION II

Free-Response 673

APPENDIX

Java Installation Guide 682

How to install Java and run a Java program? 682

Is Java free for commercial use? 686

Java Integrated Development Environments and Notebooks 687

Java Quick Reference 689

KEY FEATURES for AP Success

Course and Exam Overview

The text begins with an introductory chapter for students to set a foundation for the learning ahead. It provides an overview of the key concepts, explains how the program is structured, and emphasizes the significance of topics in the AP Computer Science A course.

WELCOME TO AP Computer Science A

Welcome to AP Computer Science A! This course is your gateway to the exciting world of programming and computational thinking. Throughout this journey, you'll learn how to solve problems using the Java programming language, develop algorithmic thinking, and create programs that manipulate and analyze data. Whether you're considering a future in software engineering, game development, data science, or simply want to enhance your problem-solving skills, this course offers valuable knowledge and experience that will serve you well in our increasingly digital world.

What is AP Computer Science A?

AP Computer Science A introduces you to computer science through programming. Unlike a general computer literacy course, AP CSA focuses on fundamental topics in computer science, including:

- **Problem-solving strategies using software design principles:** You'll learn systematic approaches to break down complex problems into manageable components, design solutions, and implement them in code.
- **Understanding and implementing algorithms:** You'll explore standard algorithms for searching, sorting, and processing data, as well as create your own algorithms for specific tasks.
- **Working with data structures to organize information:** You'll learn how to store and manipulate collections of data using arrays, ArrayLists, and 2D arrays to efficiently manage information in your programs.
- **Writing, analyzing, and debugging programs:** You'll develop skills to create working programs, understand how existing code functions, and identify and fix errors when they occur.

The course emphasizes object-oriented programming (OOP) using the Java programming language. By learning Java—a versatile, widely-used programming language—you'll develop skills that can transfer to other programming languages and environments. Java's structured, object-oriented approach provides an excellent foundation for understanding core programming concepts that apply across the field of computer science.

WELCOME TO AP Computer Science A

FOUNDATIONAL CONCEPTS in Computer Science

Introduction to Boolean Logic

What Is Boolean Logic?

Boolean logic is a mathematical system developed by George Boole in the 19th century. It is based on three main operations:

- AND (&&)
- OR (||)
- NOT (!)

In Boolean logic:

- True is represented as 1
- False is represented as 0

FOUNDATIONAL CONCEPTS in Computer Science

Debugging with Trace Tables

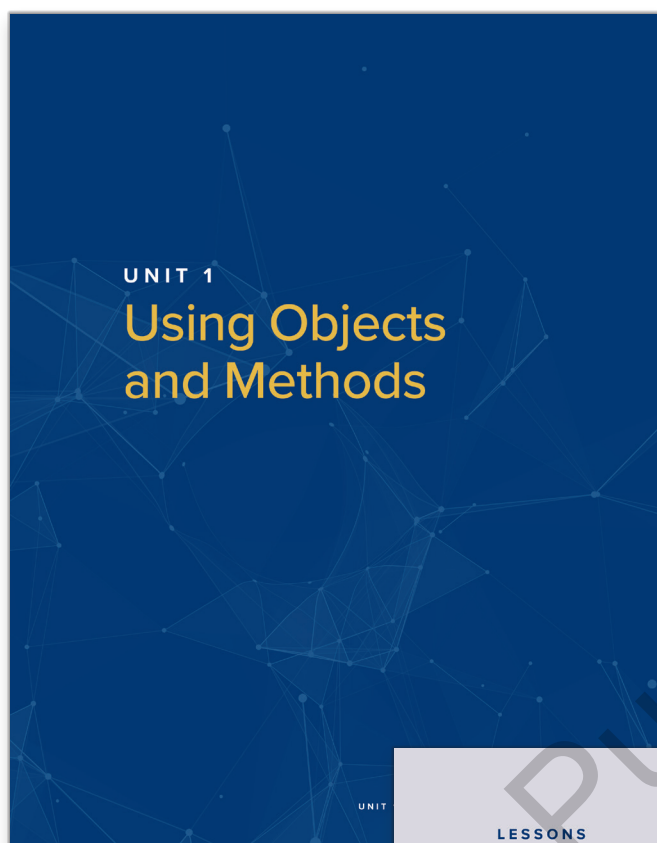
When you're learning to program, it can be hard to predict exactly how your variables change over time, especially once you introduce loops and conditionals. A trace table is a simple, "by-hand" debugging tool that forces you to step through your code one line at a time and record the values of each variable.

Setting up your trace table

1. List every variable you'll need to track (including loop counters).
2. Draw a column for each variable, plus one for the "Step" number (or the line of code).
3. Label the rows:
 - Row 0 (before the first line executes)
 - One new row for each subsequent execution step

The **Foundational Concepts in Computer Science** section provides an at-a-glance overview of essential concepts students need before beginning Java programming.

Discrete Unit Focus



Fully aligned with the AP Computer Science A framework, the textbook is organized into four units, each providing a discrete focus on core concepts required for the Exam.

The **lesson overview** provides an at-a-glance summary of the topics students will learn in each unit.

LESSONS

- [1.1 Introduction to Algorithms, Programming, and Compilers](#)
- [1.2 Variables and Data Types](#)
- [1.3 Expressions and Output](#)
- [1.4 Assignment Statements and Input](#)
- [1.5 Casting and Range of Variables](#)
- [1.6 Compound Assignment Operators](#)
- [1.7 Application Program Interface \(API\) and Libraries](#)
- [1.8 Documentation with Comments](#)
- [1.9 Method Signatures](#)
- [1.10 Calling Class Methods](#)
- [1.11 Math Class](#)
- [1.12 Objects: Instances of Classes](#)
- [1.13 Object Creation and Storage \(Instantiation\)](#)
- [1.14 Calling Instance Methods](#)
- [1.15 String Manipulation](#)

Lesson Introduction and Pages

LESSON 1

Introduction to Algorithms, Programming, and Compilers

AP Learning Objectives

- 1.1.A** Represent patterns and algorithms found in everyday life using written language or diagrams.
- 1.1.B** Explain the code compilation and execution process.
- 1.1.C** Identify types of programming errors.

Introduction to Programming and Java

Welcome to the fascinating world of programming with Java! Whether you're a complete beginner or someone with experience in other programming languages, this book is designed to guide you through the essentials of Java programming and prepare you for the AP Computer Science A exam. **Java** is a powerful, versatile, and widely-used programming language that powers everything from mobile applications to large-scale enterprise systems.

Java was first released by Sun Microsystems in 1996 (Java 1.0). The language was originally developed by James Gosling and his team as part of the Green Project, which aimed to create software for embedded devices. However, with the rise of the internet, Java quickly became popular as a language for developing web-based applications due to its ability to run seamlessly across different platforms.

Over the years, Java has evolved significantly. Oracle Corporation, and the language has undergone numerous updates, adding new features and improvements while maintaining backward compatibility. This ensures that older Java programs can still run on newer versions of the language.

The **AP Learning Objectives** clearly highlight the key concepts students will master, connecting classroom lessons directly to the goals of the AP Computer Science A framework.

Detailed **tables** illustrate core concepts, such as Boolean logic, inputs, and outputs, helping students systematically visualize programming constructs.

- Truth tables** are often used to prove the equivalency of Boolean expressions by listing all possible input combinations and comparing the results for each expression. For example the truth tables for the expressions above would be:

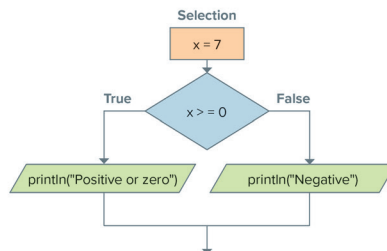
a	b	a b	(a b) == true
True	True	True	True
True	False	True	True
False	True	True	True
False	False	False	False

a	b	a == false	b == false	a == false && b == false	!(a == false && b == false)
True	True	False	False	False	True
True	False	False	True	False	True
False	True	True	False	False	True
False	False	True	True	True	False

2. De Morgan's Laws:

- De Morgan's Laws allow for converting expressions involving **AND**, **OR**, and **NOT** into alternative but equivalent forms. These transformations are particularly useful for simplifying expressions, making the code cleaner, and improving readability.

Selection is the process by which an algorithm makes choices. Based on a certain condition, the algorithm will follow one path or another. For example, after cooking pasta, you might decide whether to add more water based on the consistency of the pasta. The outcome changes depending on the situation.



204 UNIT 2 Selection and Iteration LESSON 1 Algorithms with Selection and Repetition

Clear and concise **diagrams** visually explain algorithmic processes, decision-making paths, and programming logic, making complex concepts easier to grasp.

Calculating the average

```

public class CalculateAverage {
    public static void main(String[] args) {
        ArrayList<Integer> scores = new ArrayList<>();
        scores.add(70);
        scores.add(85);
        scores.add(90);
        scores.add(65);

        int sum = 0;
        for (int score : scores) {
            sum += score;
        }

        double average = (double) sum / scores.size();
        System.out.println("Average score: " + average);
    }
}

```

Real, runnable **Java code** examples are shown in blue. Students can easily copy, paste, and execute these snippets in their programming environment to reinforce concepts.

For example:

```
System.out.println("He said, \"Hello!\");
```

Output:

```
He said, "Hello!"
```

Concise explanations accompany code examples, highlighting essential syntax and language-specific details, ensuring students clearly understand key Java programming constructs. The green highlighted text shows the **exact output** students will see when the code is run to confirm that the program is behaving as expected.

Overflow/Underflow

Assigning a value that exceeds the range of the data type.

```
int positiveLargeNumber = 2147483648;
// Overflow: Value is larger than Integer.MAX_VALUE
```

```
integer number too large
```

```
int negativeLargeNumber = -2147483649;
// Underflow: Value is smaller than Integer.MIN_VALUE
```

```
integer number too large
```

Error messages are clearly highlighted in red, instantly alerting students to common mistakes and their causes.

End-of-Lesson Review

Key Takeaways

- Informal run-time analysis helps estimate the performance of code based on statement execution counts. The **execution counts** help gauge the performance of loops and algorithms, allowing programmers to optimize and scale their code.
- By examining loop structures, you can predict how a program will scale as input size increases, allowing for more informed algorithm choices.
- Understanding different growth patterns (linear, quadratic, logarithmic) is crucial for writing efficient code that can handle large data sets without significant slowdowns.
- Nested Loops and Conditions significantly impact execution counts, and understanding these effects allows for efficient algorithm design, especially in complex applications.

Key Terms

execution count

The total number of times a statement runs within a loop or program.

linear growth

Growth rate where the execution count scales directly with the input size.

quadratic growth

Growth rate where execution count scales with the square of the input size, often due to nested loops.

logarithmic growth

Growth rate where execution count increases slowly, typically when reducing the problem size by half each iteration.

Key Takeaways summarizes the most important concepts from the lesson, providing an opportunity to review and reinforce the material before moving on.

Key Terms lists the essential vocabulary from the lesson, promoting mastery of the precise language used in the course and on the exam.

300 UNIT 2 Selection and Iteration LESSON 12 Informal Run-Time Analysis

Activities

End-of-lesson activities are designed to reinforce understanding and provide hands-on practice.

These include:

- > **Short answer questions** to review key concepts and terminology.
- > **Discussion prompts** for collaborative reflection on what students have learned.
- > **Mini-coding challenges** or **debugging exercises** that let students apply ideas immediately in actual code.

Activities

1. Write an `Employee` class with:

- A private `name` (`String`) and `salary` (`double`).
- A constructor to initialize these attributes.
- An accessor method `getSalary()`.
- A mutator method `increaseSalary(double percent)` to increase the salary by a given percentage.

2. Create a `Rectangle` class with:

- Private attributes for `width` and `height`.
- Methods to:
 - > Set the dimensions.
 - > Calculate and return the area.
 - > Calculate and return the perimeter.

3. What is the output of the following Java code:

```
public class Weather {
    private double temperature;

    public Weather(double temp) {
        temperature = temp;
    }

    public void displayTemperature() {
        System.out.println("Current Temperature: "
            + temperature + "%C");
    }
}
```

372 UNIT 3 Class Creation LESSON 6 Methods: How to Write Them

End-of-Unit Questions

Exam Questions

1. What is the correct way to define a class in Java?

- (A) `class MyClass { }`
- (B) `public class MyClass { }`
- (C) `static class MyClass { }`
- (D) Both (A) and (B)

2. Consider the following class designed to represent a Car:

```
public class Car {
    private int speed;

    public Car(int s) {
        speed = s;
    }

    public int getSpeed() {
        return speed;
    }
}
```

Now consider this code that appears in a method in another class.

```
Car myCar = new Car(60);
System.out.println(myCar.getSpeed());
```

What is the result of executing this code?

- (A) 0
- (B) 60
- (C) null
- (D) Compilation error

418 UNIT 3 — Class Creation — Exam Questions

The end-of-unit **practice questions** mimic the multiple choice format students will see on exam day, allowing them to:

- > Apply their knowledge of Java programming concepts, algorithms, and data structures in authentic contexts.
- > Become familiar with both multiple-choice and free-response question formats.
- > Evaluate their readiness by identifying mastered topics and areas requiring further review before the exam.

Full-Length Practice Exam

A complete AP-style **practice exam** is included at the back of the book. It mirrors the format of the actual test, featuring multiple-choice questions in Section I and free-response coding prompts in Section II.

This authentic practice ensures students become comfortable with exam conditions and expectations.

Section I includes multiple-choice questions that assess core Java concepts, logic, and structure. The variety helps students become familiar with the question formats used on the exam.

PRACTICE EXAM

SECTION I

Multiple Choice

1. A programmer is designing a `Book` class. Each `Book` object should store a `title` that never changes after the book is created. Which is the best way to design the `title` instance variable?

- (A) Declare `title` as `private`.
- (B) Declare `title` as `static`.
- (C) Declare `title` as `final`.
- (D) Use a setter method to update `title` later if needed.

2. Which of the following is NOT a valid way to declare an array in Java?

- (A) `int[] arr = new int[5];`
- (B) `double arr[] = {1.0, 2.0, 3.0};`
- (C) `String arr = {"A", "B", "C"};`
- (D) `boolean[] arr = new boolean[3];`

648 Practice Exam — SECTION I

SECTION II

Free-Response

Notes:

- Assume that the classes listed in the Java Quick Reference have been imported where appropriate.
- Unless otherwise noted in the question, assume that parameters in method calls are not null and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods will not receive full credit.

1. Event Scheduling

You are writing methods for a class that manages scheduling of events in a programming workshop. Each event has a start time and a duration, both represented as whole hours in 24-hour format (e.g., 13 represents 1:00 PM).

- Events must not overlap.
- All events must start and end between 8 and 18, inclusive.
- Each event lasts between 1 and 4 hours, inclusive.

The class includes a utility method `getEndTime`, which returns the end time of an event given its start time and duration.

You will write two methods to determine whether a new event can be scheduled without conflict and to find the first valid start time for a new event.

EventScheduler Class

```
public class EventScheduler {
    /* Returns the end time of an event given its
       start time and duration. */
    public int getEndTime(int startTime,
                        int duration) {
        return startTime + duration;
    }
}
```

Section II provides authentic free-response prompts. These practice questions help students clearly articulate their Java solutions and strengthen skills such as writing code, designing algorithms, and explaining inputs and outputs thoroughly.

Appendix

The **Java Installation Guide** offers students step-by-step instructions for setting up their Java development environment. Clear guidance helps students seamlessly prepare the tools they need for coding, debugging, and testing Java programs aligned with exam requirements.

APPENDIX

Java Installation Guide

How to install Java and run a Java program?

To begin, you will need a personal computer. If your school does not provide one for development purposes, Java development is not resource-intensive, so any modern computer with a decent amount of memory and storage should suffice. Java runs on Windows, macOS, and Linux, so you can use any operating system you're comfortable with. Alternatively, Chromebooks can also be used for developing and executing Java programs by utilizing online compilers.

The next element is the Java Development Kit (JDK). The JDK is a software development kit required to compile and run Java programs.

The JDK contains:

- JRE (Java Runtime Environment): The JRE is part of the JDK, allowing you to run Java applications.
- Java Compiler (javac): Converts Java source code into bytecode.
- Debugger (jdb): Used for finding and fixing bugs.
- JavaDoc: A tool to generate documentation from Java source code comments.
- Additional Tools: Includes tools like `jar` (to package Java programs into JAR files) and other utilities.

Here's a step-by-step guide on how to install the JDK on your computer.

Step 1: Download the JDK

1. Visit the Official Oracle Website:

- Go to the Oracle JDK download page (<https://www.oracle.com/java/technologies/downloads/>).
- Alternatively, you can use OpenJDK, an open-source implementation of the JDK. Visit the OpenJDK download page to find the latest version (<https://jdk.java.net/>).

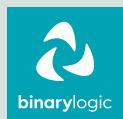
Java Quick Reference

Accessible methods from the Java library that may be included in the exam.

Class Constructors and Methods	Explanation
String Class	
String(String str)	Constructs a new String object that represents the same sequence of characters as str
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 is equal to 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
String[] split(String del)	Breaks a string into smaller substrings based on a specified delimiter del and returns these substrings as an array of String.
Integer Class	
Integer(int value)	Constructs a new Integer object that represents the specified int value
Integer.MIN_VALUE	The minimum value represented by an int or Integer
Integer.MAX_VALUE	The maximum value represented by an int or Integer
int intValue()	Returns the value of this Integer as an int
Double(double value)	Constructs a new Double object that represents the specified double value

Java Quick Reference 689

A concise **Java Quick Reference** guide is provided for students at the back of the book. It summarizes essential Java methods, constructors, and classes required for the exam.



BEST-IN-CLASS

Digital Resources

AP Computer Science Principles is enriched with multimedia content that enhances the teaching and learning experience both inside and outside the classroom.

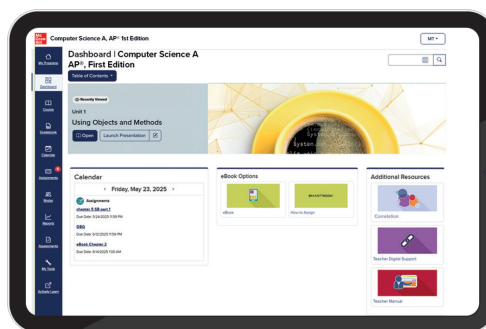
Developed with the world's leading subject matter experts and organized by unit level, the resources provide students with multiple opportunities to contextualize and apply their understanding. Teachers can save time, customize lessons, monitor student progress, and make data-driven decisions in the classroom with the flexible, easy-to-navigate instructional tools.

Student Assignments

Resources are organized at the unit level. To enhance the core content, teachers can add assignments, activities, and instructional aids to any lesson. The course gives students access to:

- an interactive eBook
- Adaptive SmartBook assignments
- two full-length AP Practice Exams
- a calendar of assignments

The course dashboard links students to resources that support success.



Teacher Resources

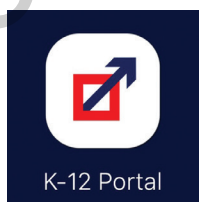
Teachers have access to the interactive eBook, plus a wealth of customizable chapter resources and powerful gradebook tools.

Resources include:

- A digital Teacher Manual with answers to the end-of-chapter questions and the Practice Exam.
- Actionable reporting features that track student progress with data-driven insights.
- A correlation to the College Board course framework, lesson plans, and additional ideas for lecture enrichment.
- On-demand digital support to help teachers navigate through all of the online features.



Powerful Gradebook reports help to inform and adjust in-class instruction.



Mobile Ready

Access to course content on-the-go is easier and more efficient than ever before with the **McGraw Hill K-12 Portal App**.



WELCOME TO AP Computer Science A

Welcome to AP Computer Science A! This course is your gateway to the exciting world of programming and computational thinking. Throughout this journey, you will learn how to solve problems using the Java programming language, develop algorithmic thinking, and create programs that manipulate and analyze data. Whether you're considering a future in software engineering, game development, data science, or simply want to enhance your problem-solving skills, this course offers valuable knowledge and experience that will serve you well in our increasingly digital world.

What is AP Computer Science A?

AP Computer Science A introduces you to computer science through programming. Unlike a general computer literacy course, the course focuses on fundamental topics in computer science, including:

- **Problem-solving strategies using software design principles:** You will learn systematic approaches to break down complex problems into manageable components, design solutions, and implement them in code.
- **Understanding and implementing algorithms:** You will explore standard algorithms for searching, sorting, and processing data, as well as create your own algorithms for specific tasks.
- **Working with data structures to organize information:** You will learn how to store and manipulate collections of data using arrays, ArrayLists, and 2D arrays to efficiently manage information in your programs.
- **Writing, analyzing, and debugging programs:** You will develop skills to create working programs, understand how existing code functions, and identify and fix errors when they occur.

The course emphasizes object-oriented programming (OOP) using the Java programming language. By learning Java—a versatile, widely-used programming language—you will develop skills that can be transferred to other programming languages and environments. Java's structured, object-oriented approach provides an excellent foundation for understanding core programming concepts that apply across the field of computer science.

Benefits of Taking AP Computer Science A

- **College Credit Opportunity:** Many colleges and universities award credit for qualifying AP scores (typically 3 or higher), potentially saving you time and money in your college education. Some institutions allow you to skip introductory computer science courses and move directly to more advanced topics.
- **Academic Challenge:** The course demonstrates your ability to handle college-level work, strengthening your college applications. Admissions officers recognize the rigor of AP courses and the dedication required to succeed in them.
- **Career Preparation:** Programming skills are valuable in many fields, from software development to data science, engineering, and beyond. Even if you don't pursue computer science as a career, the logical thinking and problem-solving skills you develop will be beneficial in almost any profession.
- **Problem-Solving Skills:** You'll develop logical thinking and analytical skills that apply across disciplines. The structured approach to problem-solving taught in computer science transfers to many other academic subjects and real-world situations.
- **Digital Literacy:** In our technology-driven world, understanding how software works gives you greater insight into the digital tools and systems you encounter daily, making you a more informed digital citizen.

Course Framework

The current AP Computer Science A course framework consolidates the content from previous versions to better align with introductory college courses and provide more in-depth coverage of essential topics. Each unit builds upon the previous one, creating a comprehensive understanding of Java programming and computer science concepts.

The Four Units

UNIT 1	Topics	Exam Weighting
Using Objects and Methods	- Variables and Data Types: Learning about primitive types (int, double, boolean) and how to store data in memory - Expressions and Operators: Using arithmetic, relational, and logical operators to manipulate data - Input and Output: Reading data from users and displaying information - Class Methods: Understanding and calling static methods like those in the Math class - Working with String Objects: Manipulating text using String methods - Object Creation: Creating instances of classes and understanding instantiation - Method Calls: Learning how to invoke instance methods on objects	15-25% of exam
UNIT 2	Topics	Exam Weighting
Selection and Iteration	- Boolean Expressions: Creating and evaluating true/false conditions - if Statements: Making decisions in code based on conditions - Nested if Statements: Implementing more complex decision structures - while Loops: Creating repetition based on a condition - for Loops: Implementing count-controlled repetition - Nested Loops: Working with loops inside other loops - Standard Algorithms: Implementing common algorithms for tasks like finding minimums/maximums, calculating sums, and counting occurrences - String Manipulation: Working with substrings, character extraction, and string processing	25%–35% of exam

UNIT 3	Topics	Exam Weighting
Class Creation	• Abstraction and Design: Planning classes to model real-world entities • Class Structure: Creating classes with attributes (instance variables) and behaviors (methods) • Constructors: Initializing objects with specific startup values • Accessor and Mutator Methods: Getting and setting object properties • Method Implementation: Writing methods that define object behaviors • Static Variables and Methods: Creating class-level elements shared across all instances • Scope and Access: Understanding variable visibility and access controls • Encapsulation: Protecting data through private variables and public methods	10%–18% of exam
UNIT 4	Topics	Exam Weighting
Data Collections	• Array Basics: Creating and accessing single-dimension arrays • ArrayList Operations: Working with dynamic, resizable collections • Array and ArrayList Traversal: Processing every element in a collection • 2D Arrays: Working with rows and columns of data • Standard Algorithms for Collections: Implementing searching, summing, counting, and filtering on data structures • Data Analysis: Processing data sets using arrays and ArrayLists • Searching Algorithms: Implementing linear and binary search • Sorting Algorithms: Understanding selection, insertion, and merge sort • Recursion: Learning about methods that call themselves • File Input: Reading data from text files	30-40% of exam

Key Changes from Previous Years

- **Text Files and Data Sets:** Additional topics on working with text files and data sets have been added to enhance real-world programming applications.
- **Consolidated Structure:** Previous Units 1-8 and 10 have been reorganized into 4 cohesive units.
- **Removal of Inheritance:** The inheritance unit (previously Unit 9) has been removed to more closely align with introductory college courses and allow deeper coverage of other topics.
- **Increased Focus on Data Collections:** With data playing an increasingly important role in computing, the revised framework places greater emphasis on data structures and algorithms for processing collections of information.
- **Exam:** Section I has 42 multiple-choice questions (increased from 40 in previous years), the weight of Section I is 55% of the total exam score (increased from 50%) and the weight of Section II is 45% of the total exam score (decreased from 50%).

The AP Computer Science A Exam

The exam tests your understanding of the course content and your ability to apply programming principles to solve problems. It consists of two sections that work together to evaluate both your conceptual knowledge and practical programming skills.

Section I: Multiple-Choice

- **Format:** 42 multiple-choice questions
- **Time:** 90 minutes
- **Weight:** 55% of total exam score
- **Question Types:**
 - > **Code Analysis Questions:** Determining what code does or identifying errors
 - > **Program Implementation Questions:** Selecting code to accomplish a task
 - > **Object-Oriented Concept Questions:** Understanding class design and behavior
 - > **Data Structure Questions:** Working with arrays, ArrayLists, and 2D arrays
 - > **Algorithm Questions:** Analyzing and implementing standard algorithms

Each multiple-choice question has 4 answer choices (decreased from 5 in previous years), which increases your chances of selecting the correct answer when making an educated guess. The questions vary in difficulty, with some requiring simple recall of facts and others demanding more complex analysis or problem-solving.

Section II: Free-Response

- **Format:** 4 questions requiring written solutions in Java
- **Time:** 90 minutes
- **Weight:** 45% of total exam score
- **Question Types:**
 1. **Methods and Control Structures (7 points):** Writing methods that use selection and iteration, including String processing
 2. **Class Design (7 points):** Creating a complete class with instance variables, constructors, and methods
 3. **Data Analysis with ArrayList (5 points):** Implementing an algorithm using an ArrayList
 4. **2D Array (6 points):** Processing a 2D array to accomplish a specific task

Each free-response question tests different skills, but all require you to write original Java code. Points are awarded for correct algorithms or code, even if minor errors exist. Clear, well-organized code is critical for earning full points.

How the Free-Response Questions are Scored

The free-response questions are scored using detailed scoring guidelines that award points for specific aspects of your solutions.

- **Algorithm Implementation:** Points awarded for implementing required algorithms correctly, showing the proper sequence of steps and logical structure to solve the problem.
- **Code Functionality:** Points awarded for code that performs the required tasks, including correct use of variables, methods, loops, conditionals, and data structures.
- **Code Organization:** Points awarded for logical structure and appropriate use of programming techniques, such as breaking down complex tasks into smaller steps.
- **Syntax and Style:** Points awarded for proper Java syntax and style conventions, including appropriate naming, indentation, and organization.

The scoring process allows you to potentially earn points even if your entire solution isn't perfect. For example, you might earn points for correctly initializing variables and setting up a loop structure, even if your algorithm has a minor logical error.

Remember that the Java Quick Reference (provided in the Appendix) will be available during the exam. It contains commonly used Java methods and classes that you may need to reference. Becoming familiar with this resource throughout the year will help you use it effectively during the exam.

How to Use This Textbook

This textbook is designed to help you master the concepts and skills you'll need for the AP Computer Science A exam and beyond. It follows the four-unit structure of the revised framework, with additional resources that support your learning journey from beginning to end.

Textbook Structure

Each unit includes:

- **Concept Explanations:** Clear, concise explanations of programming concepts with examples help you understand both the "what" and the "why" behind each topic. These explanations include analogies to everyday experiences that make abstract concepts more concrete.
- **Code Examples:** Annotated Java code samples demonstrate key concepts with detailed explanations of how the code works. These examples progress from simple to more complex implementations to gradually build your understanding.
- **Coding Activities:** Graduated challenges and hands-on programming tasks build your skills progressively, starting with basic syntax practice and building to more complex problem-solving. These activities often involve modifying existing code or creating new programs from scratch.
- **Multiple-Choice Practice:** Questions similar to those on the AP exam familiarize you with the exam requirements.
- **Free-Response Practice:** Questions modeled after the AP exam free-response questions help you practice and understand what is expected.
- **End-of-Unit Reviews:** Reviews summarize key concepts and provide a glossary of key terms.

A **full practice test** at the end of the book lets you assess your readiness under timed conditions. An additional full test is provided online for extra practice.

Study Strategies for Success

Studying for AP Computer Science A requires consistent practice and application. Unlike subjects where memorization might be sufficient, computer science demands active engagement with the material through regular programming practice. Here are some strategies to help you succeed:

Year-Round Study Tips

1. **Code Regularly:** Practice programming consistently, even if it's just for 20-30 minutes daily. Regular practice builds muscle memory for syntax and reinforces concepts better than occasional marathon sessions.
2. **Review Concepts:** After learning a new concept, review it within 24 hours to reinforce your understanding. Try to explain the concept in your own words, as teaching something (even to yourself) solidifies understanding.
3. **Create Study Groups:** Work with classmates to discuss concepts and review code together. Explaining your code to others and hearing different approaches to problems deepens your understanding.
4. **Use Multiple Resources:** Supplement this textbook with online tutorials, videos, and coding platforms.
5. **Practice Debugging:** Intentionally create and fix errors to improve your troubleshooting skills. Understanding common error messages and how to resolve them will save you time during programming assignments and exams.
6. **Connect Concepts:** Make connections between new topics and previously learned material. Understanding how concepts build upon each other creates a stronger foundation of knowledge.
7. **Track Your Progress:** Keep a log of concepts you've mastered and those you find challenging. This helps you allocate study time effectively and recognize your growth throughout the course.

Using This Textbook Effectively

1. **Read Actively:** Try to understand the "why" behind each concept, not just memorize syntax. Ask yourself questions like "Why would a programmer use this approach?" or "What problem does this solution solve?"
2. **Code Along:** Type out the examples as you read them, don't just read the code. The physical act of typing the code helps reinforce syntax and structure in your memory.
3. **Modify Examples:** Change values, add features, or combine concepts to deepen your understanding. Experimenting with code helps you see the effects of different approaches and builds confidence in your programming abilities.

4. **Complete All Exercises:** Don't skip the practice problems—they build important skills progressively and often introduce edge cases or special considerations that expand your understanding.
5. **Review Incorrect Answers:** Understand why your approach didn't work rather than just moving on. Learning from mistakes is one of the most effective ways to improve your programming skills.
6. **Use the End-of-Unit Reviews:** These summaries provide a condensed reference for key concepts and offer additional practice to reinforce your learning before advancing to the next unit.
7. **Relate to Real-World Scenarios:** Think about how the concepts you're learning might apply to real problems or applications you're interested in. This makes the material more meaningful and memorable.

Preparing for the AP Exam

1. **Start Early:** Begin exam practice at least 2-3 months before the exam. Create a study calendar to systematically cover all topics, allowing time to strengthen weak areas.
2. **Practice Timing:** Complete practice questions within the suggested timeframes to build speed and efficiency. For multiple-choice questions, aim for about 2 minutes per question; for free-response questions, budget your time based on point values.
3. **Master the Free-Response Format:** Practice writing clear, concise solutions that meet all requirements. Remember that readers look for specific elements when scoring, so make sure your code implements all requested functionality.
4. **Use the Java Quick Reference:** Familiarize yourself with this document throughout the year so you can quickly locate needed methods during the exam. Practice solving problems using only the methods included in the reference.
5. **Take Practice Exams:** Complete full-length practice exams under timed conditions to build stamina and identify areas where you need additional practice. Review both correct and incorrect answers to understand the reasoning behind each solution.
6. **Focus on Problem Types:** Become familiar with the types of problems that commonly appear on the exam. For instance, practice working with arrays, loops, and conditional statements in various combinations.
7. **Review Previous Exams:** Look at released free-response questions from previous years to understand the style and expectations of the exam. While the exact content may differ with the revised framework, the general approach to problem-solving remains similar.

Common Challenges

Understanding Object-Oriented Programming

Draw diagrams of classes, objects, and their relationships. Create analogies between programming objects and real-world objects, such as comparing a Car class to actual cars with properties (color, model) and behaviors (accelerate, brake). Build simple classes with minimal functionality first, then gradually add complexity as your understanding grows.

Tracing Complex Code

Use a trace table to track variable values as code executes. Create columns for each variable and rows for each step of execution, recording how values change. Practice step-by-step execution of code fragments regularly, writing down the state after each line executes. For nested loops or complex conditional structures, use indentation or color-coding to visually track different levels of nesting. For more details, see “Debugging with Trace Tables” in the Appendix.

Time Management on the Exam

Practice with timed exercises throughout the year. For free-response questions, start with a quick outline of your solution before coding. Allocate time based on point values—questions worth more points deserve more time. If you get stuck on a problem, make note of your approach and move on, returning later if time permits. Remember that partial solutions can earn significant points.

Syntax Errors in Java

Keep a personal log of common errors you make and review it regularly. Use consistent formatting in your code to make errors more visible. Learn to read compiler error messages effectively—they often point directly to the source of the problem. Practice correcting broken code samples to develop your debugging skills. Using an IDE with syntax highlighting can also help identify errors more quickly.

Applying Concepts to New Problems

After learning a concept, challenge yourself to apply it in different contexts. If you learn about loops for processing arrays, try using similar techniques with Strings or ArrayLists. Look for patterns across different problem types and consider how familiar solutions might adapt to new scenarios. Discuss problem-solving approaches with classmates to expose yourself to different perspectives.

Remembering Method Details

Create reference cards for frequently used methods, including their parameters and return types. Practice implementing methods from memory, then check your work against documentation. Focus on understanding the purpose of methods first, as the exact syntax can be looked up in the Java Quick Reference during the exam. Group related methods together in your study materials to reinforce connections.

A Journey of Growth

Learning to program is a journey that involves making mistakes, learning from them, and gradually building your skills. Don't be discouraged by early challenges—every experienced programmer was once a beginner. The path to proficiency involves cycles of confusion, discovery, and mastery as you encounter each new concept.

Remember these key points as you progress through the course:

- **Mistakes are valuable:** They help you identify gaps in your understanding and create memorable learning experiences. The process of finding and fixing errors develops crucial debugging skills that are essential for every programmer.
- **Persistence matters:** Some concepts may take time to sink in, and that's normal. Complex topics like recursion or object relationships often become clearer after multiple exposures and different examples. Be patient with yourself as your understanding develops.
- **Ask for help:** Utilize your teacher, classmates, and online resources when stuck. Programming is collaborative by nature, and seeking assistance is a sign of wisdom, not weakness. Often, simply explaining your problem to someone else can lead to insights.
- **Apply your knowledge:** Look for opportunities to use your programming skills in other subjects or personal projects. Creating programs that solve real problems that interest you provides motivation and reinforces learning in meaningful ways.
- **Celebrate progress:** Take note of how far you've come from your first program. Each concept you master, each problem you solve, represents growth in your capabilities as a programmer. Recognizing these achievements builds confidence and motivation.

Ready to Begin

You are now ready to embark on your AP Computer Science A journey! Before diving into Unit 1, you may want to explore the Foundational Concepts section. It offers a helpful introduction to essential computer science ideas and can strengthen your understanding before starting Java programming.

Unit 1 will introduce you to the fundamentals of Java programming and object-oriented concepts. You'll learn how to create and manipulate variables, work with expressions and operators, and begin using objects and methods—the building blocks of Java programs.

As you progress through the units, you'll develop increasingly sophisticated programming skills, from making decisions with conditional statements to creating your own classes and working with complex data structures. By the end of this course, you'll have developed valuable programming skills that can serve you in college and beyond.

The world of programming offers endless opportunities for creativity, problem-solving, and innovation. Whether you're looking to pursue computer science as a career or simply want to develop valuable technical skills, AP Computer Science A provides an excellent foundation. Your journey starts here!

For Review Purpose Only

FOUNDATIONAL CONCEPTS in Computer Science

This section introduces you to essential computer science concepts. It gives you a strong foundation before you start learning Java programming. You'll learn how computers work, how they store and use information, and how we can solve problems in a way computers understand. This will make it easier to learn programming later in the course. Don't worry if you haven't done this before – everyone can learn!

The Essence of Computer Science

What is Computer Science?

Definition and Scope

Computer science is the study of how computers process, store, and manage information. It includes both theoretical principles (such as algorithms and computation models) and practical applications (such as designing software and building systems). Computer science covers a wide range of topics, from low-level machine operations to high-level programming.

Computers have revolutionized almost every field, from healthcare to finance to entertainment. Understanding computer science means understanding the digital world, from the apps on your phone to how data is secured online.

Key Areas of Study in Computer Science

- **Algorithms & Problem-Solving:** Learning how problems can be broken down into smaller, manageable steps.
- **Software Development:** Writing programs and designing software to solve real-world problems.
- **Hardware & System Architecture:** Understanding how the different parts of a computer interact to execute tasks.
- **Data Representation & Processing:** Exploring how computers store and manipulate different types of data (like numbers, text, and pictures).

Why Study Computer Science?

Computer science is more than just programming—it's about developing logical thinking, problem-solving, and analytical skills. It helps you:

- **Solve Problems Efficiently:** Learning how to break down complex tasks into simple steps makes solutions easier to find and implement.
- **Understand Technology:** Almost every industry relies on computers. Learning computer science provides insights into how modern tools and systems function.
- **Prepare for the Future:** Many careers now require computational thinking, whether in business, science, or engineering.

Computational Thinking and Algorithms

Computational Thinking

Computational thinking is a structured way of solving problems that computers can understand and execute efficiently. It's like having a special set of tools for thinking.

The Four Key Steps

1. Decomposition

Breaking a big problem into smaller, easier-to-solve parts. When solving a complex problem, breaking it into smaller tasks makes it easier to tackle. Each part can be solved independently and then combined to form a complete solution.

Example: To develop a video game, designers split the project into different aspects such as graphics, physics, and game logic, allowing specialized teams to work on each component simultaneously.

2. Pattern Recognition

Identifying similarities in problems to apply known solutions. Recognizing repeating patterns helps solve new problems more efficiently by applying solutions that worked in similar situations. This reduces effort and improves accuracy.

Example: Recognizing traffic patterns can help city planners optimize road signals to reduce congestion. If a city experiences heavy traffic at a particular intersection every morning, planners can adjust signal timings based on observed patterns.

3. Abstraction

Ignoring unnecessary details and focusing on the core problem. Abstraction helps simplify problems by filtering out details that are not relevant to solving the main issue. This is useful in both programming and real-world problem-solving.

Example: A subway map highlights only train routes, not every street in the city. By omitting unnecessary details, it makes navigation easier for commuters.

4. Algorithm Design

Writing step-by-step instructions to solve a problem. An algorithm is a structured sequence of steps designed to perform a specific task. Writing an efficient algorithm ensures that a solution can be executed consistently and accurately.

Example: Writing a recipe with steps ensures that anyone can cook the same dish. Each step must be clear, logical, and in the correct order to achieve the desired result.

Computational thinking is not just for programmers—it is used in business, medicine, science, and engineering to break down complex problems.

Understanding Algorithms

What is an Algorithm?

An algorithm is a step-by-step set of precise instructions used to solve a problem or complete a task. It defines a structured way of processing data or making decisions, ensuring that the same problem is solved consistently and efficiently.

Why are Algorithms Important?

- They automate tasks in software and hardware.
- They optimize performance, reducing the time and resources needed to solve problems.
- They are fundamental in artificial intelligence, machine learning, data analysis, and cryptography.

Algorithms in Daily Life

- Cooking a Recipe: A recipe provides a list of ingredients (input) and step-by-step instructions (algorithm) to produce a dish (output).
- Google Search: Uses an algorithm to rank and display the most relevant results for a search query.

- Online Shopping Checkout: Ensures that the system applies discounts, calculates the total, and processes payment in the correct sequence.
- Traffic Navigation: GPS systems find the fastest route to your destination based on algorithms.
- Spam Filters: Detect unwanted emails using pattern-matching algorithms.

Computer Hardware

Core Hardware Components

A computer consists of physical hardware components that work together to process and store data.

Main Components and Their Functions

- Central Processing Unit (CPU): The "brain" of the computer that executes instructions.
- Memory (RAM): Temporary storage for information the computer is using at the moment. It's like a workbench in a workshop—tools and materials are placed there while working, but they are put away when the work is finished. When the computer is turned off, anything stored in RAM disappears, just like clearing the workbench at the end of the day.
- Storage (HDD/SSD): Long-term storage for your files, programs, and operating system. It's like a filing cabinet – you can keep things there even when the computer is off.
- Input Devices: Hardware like keyboards, mice, and touchscreens that allow users to interact with the computer.
- Output Devices: Devices like monitors, printers, and speakers that provides you with the results of the processing of data.

Hardware and software work together. For example, in a vending machine, the keypad and coin mechanism are hardware. The software checks if you've inserted enough money and then activates the motor (hardware) to give you your snack.

Variables and Data Representation

Variables and Memory

A variable is like a labeled box where the computer stores information. Example: You might have a box labeled "score" and put the number 100 in it. Or a box named "name" with the text "Maria" inside.

When a program runs, the computer sets aside space in memory for each variable.

Number Systems

Computers use different number systems to process data.

- **Binary (Base-2):** Uses only two digits: 0 and 1. Binary is used because computers store data as electrical signals, which are either ON (1) or OFF (0). This works much like a light switch: it's either on or off, with no in-between. With just these two digits, any number can be represented by combining them in different positions. For example, the decimal number 5 is represented as 101 in binary, and the binary number 1000 corresponds to the decimal number 8.
- **Decimal (Base-10):** This is the familiar number system we use every day. It is based on ten digits: 0–9.
- **Hexadecimal (Base-16):** Sometimes used in computing to represent binary numbers more compactly. It uses the digits 0–9 and the letters A–F (where A represents 10, B represents 11, up to F, which represents 15) to represent values. This makes it easier to read and write large binary numbers since each hexadecimal digit corresponds to four binary digits (bits). For example, FEED in hexadecimal is the decimal number 65,261.

ASCII and Data Encoding

Computers represent letters and other characters (text) using codes like ASCII, where each letter corresponds to a binary value. For example, the letter "A" has a specific binary code. When you type "A" on your keyboard, the computer stores it in binary using the ASCII code.

ASCII character reference table with hexadecimal and decimal values												
Dec	Hex	Char	Action(if non-printing)	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	(null)	32	20	Space	64	40	@	96	60	`
1	1	SOH	(start of heading)	33	21	!	65	41	A	97	61	a
2	2	STX	(start of text)	34	22	"	66	42	B	98	62	b
3	3	ETX	(end of text)	35	23	#	67	43	C	99	63	c
4	4	EOT	(end of transmission)	36	24	\$	68	44	D	100	64	d
5	5	ENQ	(enquiry)	37	25	%	69	45	E	101	65	e
6	6	ACK	(acknowledge)	38	26	&	70	46	F	102	66	f
7	7	BEL	(bell)	39	27	'	71	47	G	103	67	g
8	8	BS	(backspace)	40	28	(	72	48	H	104	68	h
9	9	HT	(horizontal tab)	41	29	)	73	49	I	105	69	i
10	A	LF	(NL linefeed, new line)	42	2A	*	74	4A	J	106	6A	j
11	B	VT	(vertical tab)	43	2B	+	75	4B	K	107	6B	k
12	C	FF	(NP linefeed, new page)	44	2C	,	76	4C	L	108	6C	l
13	D	CR	(carriage return)	45	2D	-	77	4D	M	109	6D	m
14	E	SO	(shift out)	46	2E	.	78	4E	N	110	6E	n
15	F	SI	(shift in)	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	(data link escape)	48	30	0	80	50	P	112	70	p

17	11	DC1	(device control 1)	49	31	1	81	51	Q	113	71	q
18	12	DC2	(device control 2)	50	32	2	82	52	R	114	72	r
19	13	DC3	(device control 3)	51	33	3	83	53	S	115	73	s
20	14	DC4	(device control 4)	52	34	4	84	54	T	116	74	t
21	15	NAK	(negative acknowledge)	53	35	5	85	55	U	117	75	u
22	16	SYN	(synchronous idle)	54	36	6	86	56	V	118	76	v
23	17	ETB	(end of trans. block)	55	37	7	87	57	W	119	77	w
24	18	CAN	(cancel)	56	38	8	88	58	X	120	78	x
25	19	EM	(end of medium)	57	39	9	89	59	Y	121	79	y
26	1A	SUB	(substitute)	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	(escape)	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	(file separator)	60	3C	<	92	5C	\	124	7C	
29	1D	GS	(group separator)	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	(record separator)	62	3E	>	94	5E	^	126	7E	~
31	1F	US	(unit separator)	63	3F	?	95	5F	_	127	7F	DEL

Other Encoding Systems

- **Unicode:** Can represent characters from almost every language in the world, including emojis.
- **RGB for Images:** Colors in images are represented using binary numbers.

Operating Systems

Operating Systems Overview

What Is an Operating System?

The operating system (OS) is responsible for managing hardware, running applications, and ensuring the system operates smoothly. It is the software that allows users and applications to interact with the hardware without needing to know the technical details.

Key Functions of an OS

1. Manages hardware resources

The operating system (OS) is responsible for controlling and coordinating the interaction between the CPU, memory, storage, and other devices to ensure the system runs smoothly. It efficiently allocates CPU time to different processes, manages memory usage, and controls input/output devices like printers, keyboards, and network connections.

Example: When you open multiple applications like a web browser, music player, and word processor, the OS ensures smooth performance by distributing CPU power among them, keeping active data in RAM for quick access, and loading files only when needed to prevent slowdowns. If you switch between programs, the OS prioritizes resources to keep the active application responsive while managing background tasks efficiently.

2. Runs software applications

The OS ensures applications run smoothly by allocating resources like CPU power, memory, and storage access. It manages multiple running programs to prevent slowdowns or crashes.

Example: When you open a music player, the OS loads it into memory, assigns CPU time for playback, and allows you to adjust settings while running other apps in the background.

3. Provides user interface (UI)

The OS offers interfaces for users to interact with the computer, either through Graphical User Interfaces (GUIs) with icons and windows or Command-Line Interfaces (CLIs) that require typed commands.

Example: Clicking an app icon in Windows or macOS launches a program instantly, while in Linux, a command like `mkdir NewFolder` creates a directory.

4. File and data management

The OS organizes files and ensures quick retrieval by maintaining a structured file system. It allows users to create, modify, and delete files efficiently.

Example: When you save a document, the OS assigns it a location, tracks it in the file directory, and ensures it can be reopened later from storage.

5. Manages security and access control

To protect system resources, the OS enforces security measures like password authentication, user permissions, and encryption to prevent unauthorized access.

Example: In Windows, user accounts ensure only authorized individuals can modify system settings, while guest users have limited access to files and applications.

Types of Operating Systems

Single-User OS: Used on personal devices like desktops and laptops: Windows, macOS, and Linux.

Multi-User OS: Supports multiple users at the same time: Unix, Linux and Windows servers.

Embedded OS: Runs on specialized hardware like ATMs, medical devices, and smart appliances: Android (for mobile devices), RTOS (real-time OS for industrial machines).

Network OS: Designed to manage network resources for multiple users: Windows Server, Linux distributions like Ubuntu or Red Hat Enterprise Server.

Process and Memory Management

The Operating System (OS) plays a crucial role in ensuring multiple programs can run efficiently without conflicts. It manages how processes are executed by the CPU and how memory is allocated to them. Without proper management, a computer could become slow, unresponsive, or even crash.

Process Management

A process is an instance of a program running on a computer. Every time you open an application, such as a web browser or a game, the OS creates a process for it and ensures it runs smoothly. The OS manages CPU time by switching between multiple processes rapidly, creating the illusion that multiple programs are running simultaneously (a technique called multitasking).

Process Lifecycle

Each process goes through a set of stages during its execution:

1. **New:** The OS creates a process when you launch a program.
2. **Ready:** The process is waiting for the CPU to execute it.
3. **Running:** The process is actively using the CPU to execute instructions.
4. **Waiting:** The process is temporarily paused, waiting for an input (e.g., waiting for a file to load or for the user to press a key).
5. **Terminated:** The process completes execution or is closed by the user.

Example: When you switch between apps on your phone or computer, the OS moves inactive apps from Running to Waiting state to free up CPU resources. When you switch back, the OS quickly restores them to the Running state. This prevents apps from consuming excessive processing power when they are not in use.

Memory Management

Since a computer has limited Random Access Memory (RAM), the OS must manage it efficiently to ensure all running applications have enough memory to function without slowing down the system. It continuously allocates and deallocates memory as processes start and end, preventing wasted space and system slowdowns.

Memory Management Techniques

1. **Paging:** Divides memory into fixed-size blocks (pages) for efficient allocation.
2. **Segmentation:** Divides memory based on logical units (e.g., functions, arrays, variables).

Example: When you open multiple tabs in a web browser, the OS assigns memory to each tab separately. If a tab becomes inactive, the OS may reduce its memory usage or suspend it to prevent unnecessary resource consumption. This prevents crashes when handling many open tabs at once.

File Systems

A file system is the method an Operating System (OS) uses to organize, store, retrieve, and manage data on a storage device such as a hard drive, SSD, USB flash drive, or memory card. It determines how files are named, stored, accessed, and managed, ensuring efficient data retrieval and storage optimization.

Every operating system relies on a file system to keep data structured and accessible. Without a file system, a storage device would be a disorganized collection of raw data, making it nearly impossible to find specific files.

Types of File Systems

1. FAT32 – Used in older storage devices and USB drives. Compatible with almost all operating systems, making it ideal for USB flash drives and external storage devices.
2. NTFS – Common in Windows systems, supports large files, file permissions, encryption, and automatic recovery features, making it more secure and efficient than FAT32.
3. ext4 – Used in Linux-based systems and optimized for performance, reliability, and large storage capacities.
4. APFS (Apple File System) – Used in macOS devices for faster file access and better encryption.

Example: When you save a Word document, the OS stores it as a data file and organizes it into a folder so you can find it later.

Device Management

The Operating System (OS) is responsible for managing input and output (I/O) devices, ensuring they function correctly and communicate properly with software applications. The OS acts as a bridge between hardware and software, handling data transfer and preventing conflicts between multiple devices.

How the OS Manages Devices

The OS manages devices using device drivers, which are specialized software that enable the OS to communicate with different hardware components. When a new device is connected, the OS installs or updates the driver to ensure compatibility.

Common I/O devices managed by the OS include:

- Keyboards and Mice – The OS ensures key presses and mouse movements are recognized and sent to applications instantly.
- Printers and Scanners – The OS queues print jobs, prioritizes them, and sends data to the printer at the correct speed.
- Storage Devices (HDDs, SSDs, USB drives) – The OS controls file read/write operations, manages storage space, and prevents data corruption.
- Monitors and Displays – The OS adjusts screen resolution, brightness, and refresh rates based on user settings and system requirements.
- Speakers and Microphones – The OS processes sound input/output, ensuring compatibility with media applications.
- Network Adapters – The OS handles Wi-Fi and Ethernet connections, managing data transfer between the computer and external networks.

Example: When you print a document, the OS places the job in a print queue to prevent conflicts if multiple applications or users send print requests simultaneously. It then sends the data to the printer at the correct speed, ensuring the document is printed accurately without errors. If any issues arise, such as low ink, paper shortages, or printer disconnection, the OS notifies the user so they can resolve the problem and complete the print job.

Software Testing and Debugging

What is Software Testing?

Software testing is the process of evaluating a program to ensure it functions correctly, meets user requirements, and operates without errors before being released. Testing helps identify bugs (errors in code), performance issues, and security vulnerabilities, preventing potential failures that could affect users. A well-tested software application is more reliable, efficient, and secure, providing a smooth user experience.

Debugging: Finding and Fixing Errors

What Is Debugging?

Debugging is the process of finding and fixing problems in a program's code.

Common Types of Bugs

1. Syntax Errors: Mistakes in the code structure.
Example: Forgetting a semicolon (;) in Java.

2. **Logic Errors:** The program runs but produces the wrong output.
Example: A program meant to add two numbers instead multiplies them.
3. **Runtime Errors:** Errors that occur while the program is running.
Example: Trying to divide a number by zero.

Debugging Techniques

1. **Print Statements:** Adding `print()` statements to check variable values.
2. **Step-by-Step Execution:** Running the program one line at a time to find errors.
3. **Breakpoints:** Pausing execution at specific lines to inspect variables.
4. **Error Logs:** Analyzing system logs to track where a crash occurred.

Types of Testing

Software testing isn't just about finding errors in code—it's about making sure a program works well, is easy to use, runs efficiently, and stays secure. Testing ensures that software applications function properly in real-world situations, just like how products are tested for quality before being sold. Below are different types of testing, explained in an easy way.

1. Unit Testing – Checking Small Parts of a Program

Imagine a math teacher checking each student's answer to a single question before grading the full test. Unit testing works the same way—it tests individual parts of a program, such as a single function or method, to make sure it produces the correct result before being combined with other parts.

2. Integration Testing – Making Sure Parts Work Together

Think of a restaurant kitchen, where different chefs cook different parts of a meal. The salad, steak, and dessert might taste great on their own, but integration testing checks if everything works well together on one plate—for example, making sure a hot dish doesn't melt a cold dessert.

3. System Testing – Testing the Entire Application

System testing is like a test drive before buying a car. Instead of checking each part separately, system testing looks at the entire program to make sure all the features work together correctly.

4. User Acceptance Testing – Does It Work for Real Users?

Even if a program works perfectly from a technical perspective, it must also be easy to use and meet users' needs. User Acceptance Testing (UAT) is like asking a group of students to test a new school app to see if they can easily find their grades, homework, and schedules without confusion.

5. Performance Testing – How Well Does It Handle Heavy Use?

Imagine a school Wi-Fi network—it works fine when only a few students are online, but what happens when everyone in the school starts streaming videos at the same time? Performance testing checks if a program can handle large amounts of users and data without slowing down or crashing.

6. Security Testing – Protecting Information from Hackers

Security testing ensures a program is safe from unauthorized access, data theft, or cyberattacks—just like how a bank secures its vaults to protect money.

Boolean Logic in Computer Science

Boolean logic is a fundamental concept in computer science that forms the basis of decision-making in programming and circuit design. It allows computers to process and evaluate conditions using true or false (1 or 0) values.

Introduction to Boolean Logic

What Is Boolean Logic?

Boolean logic is a mathematical system developed by George Boole in the 19th century. It is based on three main operations:

- AND (&&)
- OR (||)
- NOT (!)

In Boolean logic:

- True is represented as 1
- False is represented as 0

Computers use Boolean logic to make decisions in if-statements, loops, and digital circuits.

Example:

- If it is raining (True) AND I have an umbrella (True), then I can go outside (True).
- If it is raining (True) AND I do not have an umbrella (False), then I stay inside (False).

Boolean Operators

AND Operator (&&)

The AND operator returns true (1) only when both inputs are true (1). Otherwise, it returns false (0).

Input		Output
Input A	Input B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Example in Real Life:

You can watch a movie if:

1. You have a ticket (True/1)
2. You are on time (True/1)

If both are true, then you can watch the movie (True/1).

OR Operator (||)

The OR operator returns true (1) if at least one of the inputs is true (1).

Input		Output
Input A	Input B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Example in Real Life:

You can pass a test if:

1. You study well (True/1)
2. The test is easy (True/1)

If either condition is true, you pass the test (True/1).

NOT Operator (!)

The NOT operator inverts the value:

- If the input is 1 (true), it becomes 0 (false).
- If the input is 0 (false), it becomes 1 (true).

Input Output	
A	NOT A
0	1
1	0

Example in Real Life:

If the statement "I am NOT hungry" is true (1), it means "I am hungry" is false (0).

XOR (Exclusive OR) Operator (^)

The XOR operator returns true (1) only if one input is true, but not both.

Input		Output
Input A	Input B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Example in Real Life:

A light switch:

- If one switch is ON (1) and the other is OFF (0), the light is ON (1).
- If both switches are ON (1), the light is OFF (0).

Boolean Expressions and Truth Tables

A Boolean expression combines Boolean operators to create more complex conditions.

Example Expression:

(A OR B) AND NOT C

Truth Table					
A	B	C	(A OR B)	NOT C	(A OR B) AND NOT C
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	0

Think of a school hallway lighting system:

- A is a motion sensor at one end of the hallway (1 = motion detected)
- B is a motion sensor at the other end
- C is a daylight sensor (1 = it's bright outside)

The hallway lights should turn on if motion is detected at either end (A OR B) and it's not already bright from daylight (NOT C).

So the expression (A OR B) AND NOT C ensures lights only turn on when there is movement and the hallway is dark.

Boolean Logic in Computer Programming

Programming languages use Boolean logic to control if-statements, loops, and conditions.

Boolean Logic in If-Statements

```
int age = 18;
if (age >= 18) {
    System.out.println("You can vote.");
} else {
    System.out.println("You cannot vote.");
}
```

- If age is 18 or more (True), it prints "You can vote."
- If age is less than 18 (False), it prints "You cannot vote."

Boolean Logic in Loops

```
int num = 1;
while (num < 5) {
    System.out.println(num);
    num++;
}
```

- The loop runs while the condition `num < 5` is True.

Boolean Logic in Search Algorithms

Boolean logic helps computers filter and search data.

Example: Searching for emails with "urgent" OR "meeting" in the subject line.

```
if (subject.contains("urgent") || subject.
contains("meeting")) {
    System.out.println("This email is important.");
}
```

- If an email contains "urgent" OR "meeting", it gets flagged.

Boolean Logic in Digital Circuits

Logic Gates and Circuits

Computers use logic gates built from transistors to process Boolean operations.

- AND Gate: Both inputs must be 1 for the output to be 1.
- OR Gate: At least one input must be 1 for the output to be 1.
- NOT Gate: Inverts the input (1 becomes 0, 0 becomes 1).

Example: Simple Digital Circuit

A security alarm system uses Boolean logic:

- If the window is open (1) OR the door is open (1), the alarm turns ON (1).

Logic circuit:

Door Open = A, Window Open = B, Alarm = A OR B

Boolean Logic in Real-World Applications

Boolean logic is used everywhere in computing:

Application	Use of Boolean Logic
Search Engines	Filters search results (AND, OR in Google searches)
Digital Circuits	Controls how computers process data
Artificial Intelligence	Decision-making in AI algorithms
Databases	Querying data (e.g., "age > 18 AND city = 'New York'")
Game Development	Handling collision detection (if player touches obstacle, game over)

Debugging with Trace Tables

When you're learning to program, it can be hard to predict exactly how your variables change over time, especially once you introduce loops and conditionals. A trace table is a simple, "by-hand" debugging tool that forces you to step through your code one line at a time and record the values of each variable.

Setting up your trace table

1. List every variable you'll need to track (including loop counters).
2. Draw a column for each variable, plus one for the "Step" number (or the line of code).
3. Label the rows:
 - Row 0 (before the first line executes)
 - One new row for each subsequent execution step

How to fill it in

1. Initialize – In row 0, write the starting values of each variable.
2. Execute one line of code – Move to the next row, update only the columns whose values change.
3. Repeat – Continue until you reach the end of the code section you are debugging (or until you spot the bug).

Worked Example

Code

```
int x = 2;
for (int i = 0; i < 3; i++) {
    x = x + i;
}
System.out.println(x);
```

Step	i	x	Notes
0	–	2	initial values
1	0	2	enter loop (i = 0); x = 2
2	0	2	execute $x = x + i \rightarrow 2$
3	1	2	next iteration (i = 1)
4	1	3	execute $x = 2 + 1 \rightarrow 3$
5	2	3	next iteration (i = 2)
6	2	5	execute $x = 3 + 2 \rightarrow 5$
7	3	5	loop ends (i = 3 $\rightarrow$ exit)
–	–	5	prints 5

Step 1: Initialize i and x.

Steps 2–6: Each time through the loop, record the updated values.

Step 7: When i reaches 3, the loop condition fails—stop tracing.

Practice

Simple conditional

Draw a trace table and find the printed result.

```
int a = 4, b = 7;
if (a < b) {
    a = a * 2;
} else {
    b = b - 3;
}
System.out.println(a + b);
```

Nested loop

Show each step of i, j, and sum.

```
int sum = 0;
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        sum += i * j;
    }
}
System.out.println(sum);
```

Mixed loop + conditional

Trace how count changes and state the final output.

```
int count = 0;
for (int n = 0; n < 5; n++) {
    if (n % 2 == 0) {
        count++;
    }
}
System.out.println(count);
```

Tips for Effective Tracing

Only record a new row when a variable actually changes, unless you're stepping through every single line.

Use trace tables to spot off-by-one errors in loop bounds or incorrect updates in conditionals.

Ready to Begin Unit 1

With these foundational concepts and tracing strategies in place, you're now ready to begin Unit 1 and start writing your first Java programs with confidence.

For Review Purpose Only

UNIT 1

Using Objects and Methods

LESSONS

1.1 Introduction to Algorithms, Programming, and Compilers

1.2 Variables and Data Types

1.3 Expressions and Output

1.4 Assignment Statements and Input

1.5 Casting and Range of Variables

1.6 Compound Assignment Operators

1.7 Application Program Interface (API) and Libraries

1.8 Documentation with Comments

1.9 Method Signatures

1.10 Calling Class Methods

1.11 Math Class

1.12 Objects: Instances of Classes

1.13 Object Creation and Storage (Instantiation)

1.14 Calling Instance Methods

1.15 String Manipulation

Introduction

This unit introduces the core principles of the Java programming language, focusing on primitive data types and their role in creating efficient programs. We will begin by exploring how to write the main method, call existing methods, handle inputs, and generate outputs. Along the way, we'll dive into three fundamental data types, learning how to declare variables, store values, and manipulate them using basic operations.

A key aspect of programming is the ability to craft mathematical expressions that model real-world quantities and variability. In Java, this requires an understanding of how specific operators function within formulas and expressions. Through mastering these concepts, we will be able to write meaningful code that reflects real-world scenarios.

Programming is inherently iterative, and mistakes are inevitable. Learning to recognize and correct errors is crucial. Exposure to compiler error messages will help you develop debugging techniques. Furthermore, the ability to explain why a code segment fails to compile or execute correctly will not only sharpen your problem-solving skills but also improve your collaboration with others. By mastering these foundational concepts, you'll be better equipped to build programs that accomplish specific tasks and pave the way for more advanced programming topics in future chapters.

LESSON 1

Introduction to Algorithms, Programming, and Compilers

AP Learning Objectives

1.1.A Represent patterns and algorithms found in everyday life using written language or diagrams.

1.1.B Explain the code compilation and execution process.

1.1.C Identify types of programming errors.

Introduction to Programming and Java

Welcome to the fascinating world of programming with Java! Whether you're a complete beginner or someone with experience in other programming languages, this book is designed to guide you through the essentials of Java programming and prepare you for the AP Computer Science A exam. **Java** is a powerful, versatile, and widely-used programming language that powers everything from mobile applications to large-scale enterprise systems.

Java was first released by Sun Microsystems in 1996 (Java 1.0). The language was originally developed by James Gosling and his team as part of the Green Project, which aimed to create software for embedded devices. However, with the rise of the internet, Java quickly became popular as a language for developing web-based applications due to its ability to run seamlessly on different systems.

Over the years, Java has evolved significantly. It is now maintained by Oracle Corporation, and the language has undergone numerous updates, each adding new features and improvements while maintaining backward compatibility — meaning older Java programs can still run on newer versions without changes.

Algorithms and the Purpose of Programming

Programming is the process of giving instructions to a computer to solve a problem or accomplish a specific task. These instructions, or algorithms, are written in a language the computer can understand and execute. Computers are exceptionally powerful, but they cannot perform any task without being explicitly told how to do it. This is where programming comes in—enabling computers to perform tasks from simple calculations to complex data processing by following a series of logical steps.

An **algorithm** is a set of ordered instructions that a program follows to solve a problem or perform a task. In programming, the proper sequencing of these instructions is essential. In a baking recipe, you wouldn't bake the cake before mixing the ingredients. Similarly, in programming, the correct order of operations ensures that the program works as expected.

Example:

Here's a simple example of an algorithm written in everyday language:

Algorithm: Making a cup of tea

- Step 1: Boil water.
- Step 2: Place a tea bag in a cup.
- Step 3: Pour boiling water into the cup.
- Step 4: Steep the tea for 3 minutes.
- Step 5: Remove the tea bag and enjoy.

In a programming context, this algorithm could be translated into a set of instructions in Java, and each step must be completed in a specific order. This process is called **sequencing**. In Java, code is executed line by line in the order it appears, unless control flow mechanisms (instructions that change the normal flow of the program, like deciding or repeating), such as loops or conditional statements (which we'll explore in later lessons), change this sequence. Algorithms can be represented using written language or diagrams, which provide a visual or textual representation of the step-by-step process involved in solving a problem.

Patterns in Programming and Everyday Life

Patterns play a key role in programming by helping identify repeated steps or logical structures that can be reused to solve problems more efficiently. In everyday life, patterns appear in routines, problem-solving strategies, and decision-making processes—like following a recipe or planning a route. Similarly, in programming, recognizing and applying patterns makes it easier to design clear, reliable algorithms.

Why Learn Java?

Java is a popular and widely-used programming language, ideal for both beginners and professionals. Its prominence in software development, particularly for creating applications that can be used on different types of devices. Java remains one of the most popular programming languages in the world. Here are several key reasons to learn Java:

- **Object-Oriented Language:** Java is fully object-oriented, meaning it organizes code into objects that encapsulate data and behaviors, allowing programs to model real-world entities and their interactions. This helps in building programs that are modular, reusable, and scalable.
- **Platform Independence:** One of Java's key advantages is its platform independence. The phrase “Write Once, Run Anywhere” is commonly associated with Java, meaning you can write Java code on one machine and run it on any other machine with the Java Virtual Machine (JVM for short, a program that runs compiled Java code and allows it to work on different types of computers) installed.
- **Extensive Libraries:** Java provides a robust collection of standard libraries, which makes it easier for developers to implement functionality without writing code from scratch.
- **Robustness and Security:** Java emphasizes strong type checking (checks that occur while the program is running, such as detecting invalid operations or accessing memory incorrectly), runtime checking, and early error detection, reducing error-prone situations. Strong type checking ensures that variables are used only with compatible data types, catching potential errors during compilation and improving program reliability.
- **Large Community and Ecosystem:** Java has a vast community of developers and a rich ecosystem of tools, frameworks, and libraries.

Some of the ideas introduced here may seem complex or unfamiliar at first, but as you work through this course, you will gradually build the knowledge needed to understand them and appreciate how they fit into programming as a whole.

How are Java programs executed?

A Java program goes through several stages from writing to execution. At first, code written in Java is transformed into an executable form using a compiler that also checks the code for certain types of errors.

Writing the Code

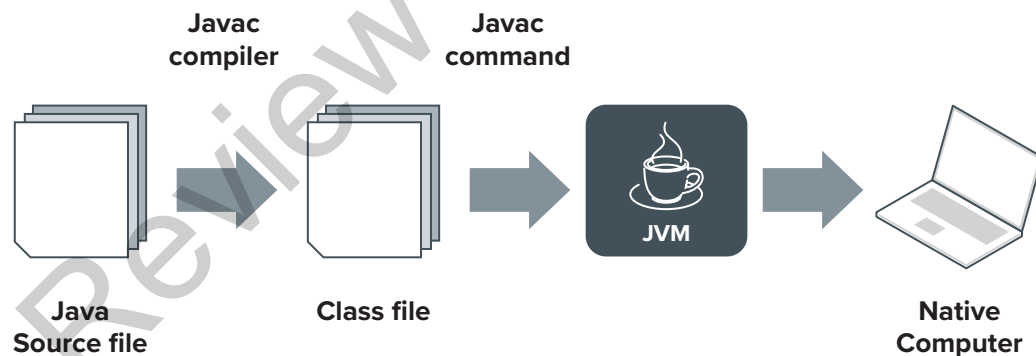
Java programs are written in text files with the extension `.java`. The programmer uses an Integrated Development Environment (IDE) to write and organize the code efficiently. Code can be written in any text editor; however, an Integrated Development Environment (IDE) is often used to write programs because it offers helpful tools, like syntax highlighting, code suggestions, and debugging features.

Compiling the Code

After writing the program (`.java` file), the next step is compiling (the process of translating human-readable code into a form the computer can understand before running it). Java code is compiled into bytecode (`.class` file), an intermediate language understood by the Java Virtual Machine (JVM). The JVM makes Java programs platform-independent by executing platform-neutral bytecode, which is generated during compilation, and dynamically translating it into machine-specific code for the underlying operating system and hardware, allowing the same program to run on any device with a compatible JVM. The compiler checks for syntax errors—errors where the code doesn't follow the language rules—and ensures the program can run correctly.

Running the Program

Once compiled with Javac (Java's compiler), the bytecode is executed by the JVM, which converts it into machine-level code that the computer can understand.



Basic Structure

The basic structure of a Java program consists of several key components, which work together to allow the program to run correctly.

Let's consider the following Java program.

Example:

```
// This is a basic Java program
public class HelloWorld {
    public static void main(String[] args) {
        /* Print a message to the console
           the phrase Hello, World! */
        System.out.println("Hello, world!");
    }
}
```

Output:

```
Hello, world!
```

Class Declaration

Every Java program is organized into classes. A class serves as a blueprint for creating objects and defines the structure of the program. The class name should match the file name (e.g., if the file is `HelloWorld.java`, the class should be named `HelloWorld`).

```
public class HelloWorld {
    // Class body
}
```

Comments

Comments are non-executable parts of the program that help explain what the code does and improve readability for both the original programmer and others. Java supports single-line comments (using `//`), multi-line comments (using `/* ... */`), and Javadoc comments (using `/** ... */`), which are specifically used to generate API documentation (officially formatted descriptions of classes and other code elements that help developers understand how to use them). Comments are ignored by the compiler and do not affect the program's execution.

```
// This is a single-line comment
/* This is a multi-line comment
   that spans multiple lines */
```

Main Method

The `main` method is the entry point of every Java program — it's where the program starts executing. The method is defined as `public static void main(String[] args)` and must be included in any executable Java program. Since all Java code must be part of a class, the `main` method is always written inside a class definition.

```
public static void main(String[] args) {  
    // Code to be executed  
}
```

Braces { }

Java uses curly braces to define the scope of classes, methods, and code blocks. Everything inside a class or method must be enclosed in these braces.

```
public class HelloWorld {  
    public static void main(String[] args) {  
        // Code block inside main method  
    }  
}
```

Semicolons ;

Every statement (a complete instruction that tells the program to do something) in Java must end with a semicolon. This signals the end of a command for the compiler.

Statements and Expressions

Inside the main method (or other methods), you write the actual code that performs the program's tasks. This includes statements and expressions, such as variable declarations (introductions of variables and their data types), calculations (mathematical operations to compute values), method calls (instructions that execute a specific block of reusable code), and control structures like loops (structures that repeat actions) and conditionals (structures that make decisions based on conditions).

```
public class HelloWorld {  
    public static void main(String[] args) {  
        // Method call: print message  
        System.out.println("Hello, World!");  
    }  
}
```

Types of Programming Errors

When programming, it's essential to be aware of the different types of errors that can occur. Understanding these errors is crucial for debugging (the process of finding and fixing errors or unexpected behavior in a program) and ensuring the smooth execution of your code.

Syntax Errors:

These occur when the rules of the language are not followed. For example, forgetting a semicolon or misspelling a keyword. These errors are caught during compilation and must be fixed before the program can run.

Logic Errors:

Errors where the program runs but produces incorrect results due to flaws in the algorithm. These are the most difficult to identify because the program runs without crashing but produces incorrect results. For example, if you intend to add two numbers but instead multiply them, the program will still run, but the result will be wrong.

Runtime Errors:

These occur during the execution of a program, such as dividing by zero. They cause the program to crash unless handled appropriately using exception handling techniques (ways to detect and respond to errors while a program is running). The compiler does not detect runtime errors — they only appear while the program is running.

Exceptions:

A specific type of runtime error that occurs during the execution of a program and interrupts the normal flow. For example, trying to access a non-existent file.

Activities

1. What is an algorithm, and why is it important in programming?
2. How does sequencing impact the correctness of an algorithm?
3. What is the purpose of programming languages?
4. What are the key stages of executing a Java program?
5. What are the three types of programming errors, and how do they differ?
6. Explain what object-oriented languages are and how Java implements object-oriented code structure and programming.
7. Create a Java program that when run, it prints in the console the message “This is my first Java program.”
8. Research and compare Java to another programming language (e.g., Python, C++).

Identify:

- 4 syntax differences
- their key features and strengths
- their popular use cases

Key Takeaways

- Programming involves solving problems by providing a computer with a structured set of instructions to perform a task.
- Java is a versatile, object-oriented programming language with cross-platform compatibility through the use of the Java Virtual Machine (JVM).
- Knowing the compilation process can help you understand how Java runs across platforms: code is written, compiled into bytecode, and then executed by the JVM.
- Common programming errors include syntax errors, runtime errors, and logic errors.
- Algorithms and the correct sequence of steps are fundamental to ensuring programs function as intended.

Key Terms

programming

Writing instructions for a computer to execute

Java

A versatile, object-oriented programming language.

algorithm

A step-by-step process for solving a problem.

sequencing

The specific order in which steps in a process are completed.

compilation

The process of translating high-level code into bytecode.

syntax error

A mistake in the structure of a program.

logic error

A flaw in the algorithm that leads to incorrect results.

runtime error

An error that occurs while the program is running.

LESSON 2

Variables and Data Types

AP Learning Objectives

1.2.A Identify the most appropriate data type category for a particular specification.

1.2.B Develop code to declare variables to store numbers and Boolean values.

In this lesson, we will explore the essential concepts of variables and data types in Java, which are essential for storing and manipulating data. Understanding how to declare variables and select the correct data types is crucial for writing flexible and reusable programs.

Understanding Variables

A **variable** is a named location in memory where data is stored. Variables allow programmers to store values and modify them throughout the program. Each variable is associated with a data type, which determines the kind of data it can hold (e.g., numbers, text, or true/false values).

Example 1 (Integer variable definition):

```
int age = 25; /* A variable named 'age' of type int
               holding the value 25 */
```

In example 1, `int` is the data type, `age` is the variable name, and 25 is the value stored in the variable. Printing (displaying a value in the console) the variable `age` will output:

```
25
```

Primitive Data Types

In Java, data types are classified into two main categories:

1. Primitive Data Types
2. Reference Data Types (covered in later lessons)

This lesson focuses on **primitive data types**, which store basic values directly in memory. While Java provides eight primitive data types, the AP Computer Science A course emphasizes three commonly used ones: `int`, `double`, and `boolean`.

Primitive data types (like `int`, `double`, and `boolean`) store fundamental values in programming, such as numbers and logical values. In contrast, reference data types (such as arrays or objects) store references to memory locations where the actual data is stored. Understanding the distinction between primitive and reference types is important for designing efficient programs.

`int` (Integer)

Purpose:

Stores whole numbers without decimals.

For example:

```
| int count = 100;
```

Size

32-bit signed integer.

Range:

-2,147,483,648 (`Integer.MIN_VALUE`) to 2,147,483,647 (`Integer.MAX_VALUE`).

`double` (Floating-point Number)

Purpose

Stores real numbers with decimal points.

For example:

```
| double price = 19.99;
```

Size

64-bit double-precision floating-point.

Range

Approximately $\pm 1.7E308$ (very large positive and negative numbers).

boolean (True/False)

Purpose

Stores logical values, either `true` or `false`.

For example:

```
| boolean isActive = false;
```

Size

Typically stored in 1 bit.

Declaring Variables

In Java, you must declare a variable before using it. Declaring a variable requires specifying the data type and providing a variable name. Optionally, you can assign an initial value during declaration. The general syntax is:

```
| dataType variableName = value;
```

Example 2 (Declaring variables):

```
| int age = 18; // Declaring and initializing an integer
| double temperature = 98.6;
| // Declaring and initializing a double
| boolean isJavaFun = true;
| // Declaring and initializing a boolean
```

Printing the variable `age` will output:

```
18
```

Printing the variable `temperature` will output:

```
98.6
```

Printing the variable `isJavaFun` will output:

```
true
```

You can also declare a variable without assigning a value immediately:

```
int year;  
year = 2024; // Variable assigned a value later
```

Printing the variable year will output:

```
2024
```

Variable Naming Rules

In Java, variable names must follow certain rules:

1. Must begin with a letter, the dollar sign (\$), or an underscore (_).
2. Cannot start with a number.
3. Cannot be a reserved keyword (e.g., `int`, `public`, `class`).
4. Case-sensitive: `myVariable` and `MyVariable` are considered different.
5. Cannot contain whitespaces (spaces, tabs, etc.) or special characters other than \$ and _.
6. Use meaningful names: It is good practice to name your variables descriptively.

Examples of valid and invalid variable names:

- Valid: `totalAmount`, `_value`, `isDone`
- Invalid: `2ndPlace`, `public`, `#flag`

Using Variables in Expressions

Variables can be used in expressions to perform operations such as addition, subtraction, multiplication, and division. The value of a variable can change throughout the program, making the code adaptable to various inputs.

Example 3 (Using variables in expressions):

```
int a = 5;  
int b = 10;  
int sum = a + b;  
// The sum of 5 and 10 is stored in the 'sum' variable
```

Printing the variable sum will output

15

In this example, the values of a and b are used to compute the sum, which is stored in the `sum` variable.

Choosing the Right Data Type and Errors to Watch Out For

Choosing the appropriate data type for a variable is crucial for program efficiency and error avoidance:

- Use `int` for whole numbers (e.g., age, count of items).
- Use `double` for numbers with decimal values (e.g., temperature, price).
- Use `boolean` for true/false values (e.g., user login status).

Example 4 (choosing data type):

```
double radius = 7.5;
/* Double is appropriate because radius can have decimal
   values */
boolean isComplete = false;
// Boolean is appropriate for a true/false value
```

Common errors

When working with variables and data types, several common errors can arise:

Type Mismatch

Assigning a value of one data type to a variable of a different type.

```
int x = 2.0; // Error: 2.0 is a double, not an int
```

incompatible types: possible lossy conversion from double to int

Overflow/Underflow

Assigning a value that exceeds the range of the data type.

```
int positiveLargeNumber = 2147483648;  
// Overflow: Value is larger than Integer.MAX_VALUE
```

integer number too large

```
int negativeLargeNumber = -2147483649;  
// Underflow: Value is smaller than Integer.MIN_VALUE
```

integer number too large

Activities

1. What is the difference between primitive and reference data types in Java?
2. Why is choosing the correct data type important in Java?
3. What is overflow and underflow in the context of data types?
4. Declare three variables to represent your age, height, and whether or not you are a student.
5. Write the following program and explain why the result appears as it does:

```
public class Test {  
    public static void main(String[] args) {  
        int maxNum = Integer.MAX_VALUE;  
        System.out.println(maxNum + 1);  
    }  
}
```

6. Write the following program and explain why the result appears as it does:

```
public class Test {  
    public static void main(String[] args) {  
        int value = 12.5;  
        System.out.println("Value: " + value);  
    }  
}
```

7. Write the following program and explain why the result appears as it does:

```
public class Test {  
    public static void main(String[] args) {  
        int num = 0; // Initialize num  
        System.out.println("Number: " + num);  
        num = 42;  
    }  
}
```

Key Takeaways

- Variables store data values in a program. Each variable has a name, a data type, and a value that can change during execution.
- Java provides several primitive data types, but for the AP Computer Science A course, we focus on:
 - > `int`: For storing whole numbers (e.g., `int age = 20;`).
 - > `double`: For storing decimal numbers (e.g., `double price = 19.99;`).
 - > `boolean`: For storing true/false values (e.g., `boolean isActive = true;`).
- Variables are declared by specifying a data type followed by the variable's name. Optionally, you can initialize the variable with a value at the time of declaration.
- Variables can be used in expressions to perform operations like addition or comparison. The values they store can be updated during the program's execution.
- It's important to choose the appropriate data type based on the kind of data you need to store—use `int` for whole numbers, `double` for numbers with decimals, and `boolean` for logical values.
- **Common Errors:**
 - > **Type Mismatch:** Attempting to assign a value of one type to a variable of another (e.g., assigning a string to an `int`).
 - > **Overflow/Underflow:** Assigning a value outside the range of a data type (e.g., a number too large for an `int`).
- Using variables allows programs to handle different inputs, making the code more flexible and adaptable to various tasks.

Key Terms

variable

A named storage location for data.

primitive data type

A simple data type that stores values directly.

int

A primitive type for whole numbers.

double

A primitive type for real numbers (decimals).

boolean

A primitive type for true/false values.

LESSON 3

Expressions and Output

AP Learning Objectives

1.3.A Develop code to generate output and determine the result that would be displayed.

1.3.B Develop code to utilize string literals and determine the result of using string literals.

1.3.C Develop code for arithmetic expressions and determine the result of these expressions.

In this lesson, we delve into expressions and their use in Java for performing calculations, manipulating data, and generating meaningful output. Understanding how expressions work and how to display output are essential skills that will enable you to write interactive programs and perform computations effectively.

Expressions are the backbone of all programs, allowing for dynamic operations and interactions. By mastering arithmetic operators and output methods, you can develop Java programs that carry out complex tasks and communicate results to users. As you continue practicing, your ability to write and debug expressions will improve, helping you solve real-world programming problems more efficiently.

Working with Java's core output methods

In Java, two primary methods are used for displaying output to the console:

- **`System.out.print`**: Prints the provided text or value but does not move the cursor to a new line afterward.
- **`System.out.println`**: Prints the text or value and moves the cursor to the next line after displaying the output.

These methods are crucial for debugging, user interaction, and presenting results.

System.out.print vs. System.out.println

Example 5 (System.out.print outputs text but keeps the cursor on the same line)

```
System.out.print("Hello ");  
System.out.print("World!");
```

Output:

Hello World!

Example 6 (System.out.println outputs text and moves the cursor to a new line)

```
System.out.println("Hello");  
System.out.println("World!");
```

ln stands for (new) line

Output:

Hello
World!

By using pre-tested, reusable code (like these output methods), you can:

- **Save time:** Avoid rewriting common functions, such as printing output.
- **Reduce errors:** These methods are thoroughly tested and reliable.
- **Focus on logic:** Instead of dealing with basic functionality, you can concentrate on solving complex problems and developing efficient algorithms.

Evaluating and Debugging Code Structures

As a beginner, learning how to evaluate and debug code is essential to improving your programming skills. Below are some common debugging tips when working with Java's output methods.

Common Mistakes and Debugging Tips

Missing Semicolons:

Java statements must end with a semicolon. Forgetting a semicolon will result in a syntax error.

Example 7 (Error missing semicolon):

```
System.out.println("Hello"); System.out.println("World!");
```

Output:

```
Hello  
World!
```

```
| System.out.println("Hello") System.out.println("World!");
```

Output:

```
',' expected
```

Note: When using tools like Jupyter Notebook (an alternative environment for Java "programming"), single-line statements may not require semicolons (as shown below), but it's good practice to include them in typical Java development environments.

```
| System.out.println("Hello, World")
```

Output:

```
Hello, World!
```

Incorrect Use of System.out.print and System.out.println:

Misusing `print` when a new line is required can lead to formatting issues.

Example 8 (improper formatting):

```
| System.out.print("Hello");  
| System.out.print("World");
```

Output:

```
HelloWorld
```

Solution: Use println to move to a new line.

```
| System.out.println("Hello");  
| System.out.println("World");
```

Output:

```
Hello  
World
```

Expressions in Java

An expression in Java is a combination of variables, operators, and values that produces a result. Expressions enable data manipulation in programs.

For example,

```
int sum = 5 + 10;  
/* 5 + 10 is evaluated to 15 and stored in the 'sum'  
   variable */
```

Here, `5 + 10` is an expression that uses numeric values and the addition operator to calculate a result, which is then stored in the `sum` variable.

Arithmetic Operators

Java supports a variety of **arithmetic operators** to perform calculations:

Addition (+)

```
int result = 10 + 5; // result is 15
```

Subtraction (-)

```
int difference = 10 - 5; // difference is 5
```

Multiplication (*)

```
int product = 10 * 5; // product is 50
```

Division (/)

When dividing integers, only the quotient is returned, and any fractional part is truncated, meaning it is discarded without rounding. The result is always an integer and does not represent the complete mathematical outcome if there is a remainder.

```
int quotient = 10 / 3; // quotient is 3
```

If either operand is a double, the division is floating-point, and the result includes the decimal part.

```
double result = 10.0 / 3; // result is 3.333...
```

Modulo (%)

The modulo operator returns the remainder after division and should only be used with integer values.

```
| int remainder = 10 % 3; // remainder is 1
```

Operator Precedence and Associativity

When multiple operators are used in an expression, Java follows a specific order called **operator precedence**, which is similar to mathematical order (PEMDAS).

Operator Precedence:

1. Multiplication (*), Division (/), Modulo (%)
2. Addition (+), Subtraction (-)

Operators with the same precedence are evaluated from **left to right**.

For example,

```
| int result = 2 * 3 + 5;  
| /* The result is 11, as multiplication is done before  
|    addition */
```

To change the order, parentheses can be used:

```
| int result = 2 * (3 + 5);  
| // The result is 16, as addition is done first
```

When negative numbers are involved, operator precedence still applies, but parentheses become especially important to avoid mistakes.

```
| int result = -2 * 3 + 5;  
| /* The result is -1, because this is evaluated as  
|    (-2 * 3) + 5 */
```

If instead you want to negate the entire expression after multiplication:

```
| int result = -(2 * 3 + 5); // The result is -11
```

The placement of the minus sign and parentheses can significantly affect the outcome, so careful use is important when working with negative values.

String Literals and Escape Sequences

A **string literal** is a sequence of characters enclosed in double quotes. Special characters can be included using escape sequences, which are sequences of characters starting with backslash (\).

Common Escape Sequences:

- Newline (\n): Moves the cursor to the next line.
- Double Quote (\"): Inserts a double quote in the string.
- Backslash (\\): Inserts a backslash character.

For example:

```
| System.out.println("He said, \"Hello!\"");
```

Output:

```
He said, "Hello!"
```

```
| System.out.println("Line1\nLine2");
```

Output:

```
Line1
Line2
```

```
| System.out.println("C:\\PathName\\File.txt");
```

Output:

```
C:\PathName\File.txt
```

Compound Expressions

Compound expressions involve multiple operators. Understanding how to evaluate these expressions requires knowledge of operator precedence (the rules that determine which operations are performed first) and associativity (the rules that determine the order in which operators of the same precedence are evaluated).

For example,

```
| int result = (4 + 3) * 2 - 1; // Result is 13
```

In this example, parentheses affect the normal precedence, ensuring that addition occurs before multiplication and subtraction.

Errors in Expressions

Common errors in expressions include:

Syntax Errors:

Mistakes like missing semicolons or incomplete expressions are caught by the compiler.

```
| int sum = 5 + ; // Error: incomplete expression
```

Arithmetic Errors

Runtime errors such as division by zero result in an `ArithmeticException`.

```
| int result = 5 / 0; // Error: Division by zero
```

Activities

1. What is the difference between `System.out.print` and `System.out.println`?
2. What is operator precedence, and why is it important?
3. What happens when you divide two integers in Java?
4. Why are expressions important in Java programming?
5. What are common mistakes when writing expressions in Java?
6. What is the value of the `result` variable?

```
int result = (5 + 3 * 2) - (4 / 2 + 6) * (3 - 1) + 8 % 3;
```

7. Write a Java program that calculates and prints the result of the following operations:
 1. The sum of two integers: `15 + 27`
 2. The product of two integers: `7 * 8`
 3. The result of dividing two integers: `100 / 9` (observe how integer division works)
 4. The remainder when dividing 100 by 9 (using the modulo operator `%`)

Example Output:

```
Sum: 42
Product: 56
Quotient: 11
Remainder: 1
```

8. Complete the following Java code so that the mean of the variables is displayed:

```
public class MeanCalculator {  
    public static void main(String[] args) {  
        // Given five integers  
        int num1 = 8;  
        int num2 = 15;  
        int num3 = 10;  
        int num4 = 20;  
        int num5 = 17;  
  
        // Compute the sum  
        int sum = _____;  
        // (A) COMPLETE THIS PART TO ADD ALL NUMBERS  
  
        // Compute the mean (average)  
        _____ mean = sum / _____;  
        // (B) COMPLETE THIS PART TO ENSURE DECIMAL PRECISION  
  
        // Display the result  
        System.out.println("Mean: " + mean);  
    }  
}
```

Key Takeaways

- Expressions are combinations of values, variables, and operators that produce a result.
- Arithmetic operators are used to perform basic mathematical operations.
- Operator precedence defines the order in which operations are evaluated.
- `System.out.print` and `System.out.println` are essential for communicating with the user.

Key Terms

expression

A combination of variables, operators, and values that Java evaluates to produce a result.

Example: `5 + 3` is an expression that evaluates to 8.

arithmetic operators

Symbols used to perform basic mathematical operations, including:

- **Addition (+):** Adds two numbers.
- **Subtraction (-):** Subtracts one number from another.
- **Multiplication (*):** Multiplies two numbers.
- **Division (/):** Divides one number by another.
- **Modulo (%):** Returns the remainder after division.

operator precedence

The rules that determine the order in which operators in an expression are evaluated. Multiplication, division, and modulo have higher precedence than addition and subtraction.

associativity

Defines the direction in which an expression with operators of the same precedence is evaluated (usually left-to-right in Java).

string literal

A sequence of characters enclosed in double quotes that represents text in Java.

Example: `"Hello, World!"` is a string literal.

escape sequence

A character combination that starts with a backslash (\) and allows the inclusion of special characters in strings.

Examples include:

- `\n`: Newline (moves the cursor to the next line).
- `\"`: Inserts a double quote.
- `\\`: Inserts a backslash.

syntax error

A mistake in the structure of a program (e.g., missing a semicolon or using incorrect operators) that prevents the program from compiling.

arithmetic exception

A runtime error that occurs during an invalid arithmetic operation, such as dividing by zero.

LESSON 4

Assignment Statements and Input

AP Learning Objectives

1.4.A Develop code for assignment statements with expressions and determine the value that is stored in the variable as a result of these statements.

1.4.B Develop code to read input.

In Java, **assignment statements** are used to assign values to variables, which must be initialized with a value before they can be used in expressions. Variables are initialized the first time they are assigned a value, and the assigned value must be compatible with the variable's data type. For reference types — which hold a reference to an object in memory rather than storing the actual data (e.g., strings) — variables can be assigned an object or the special **null** value to indicate that no object is associated. The Scanner class is used to make programs interactive by reading input from the user, such as integers, floating-point numbers, and strings. Input validation is crucial to prevent errors from incompatible data, and Java provides mechanisms like try-catch blocks to handle these cases gracefully.

Assignment Statements

An **assignment statement** is used to assign a value to a variable. It involves the assignment operator = which links the variable on the left to the value or expression on the right. A variable must be assigned a value **before** it can be used in an expression. This process is called **initialization**, and it must happen the first time a variable is assigned a value.

The value assigned must be compatible with the variable's data type. For example, an int variable can only be assigned integer values.

For example:

```
int age = 25; /* Assigns the value 25 to the integer
              variable 'age' */
```

In this example, `int` is the data type, `age` is the variable, and `25` is the value being assigned.

Initializing and Updating Variables

When a variable is assigned a value for the first time, it is **initialized**. Variables can later be updated by assigning new values to them.

Example:

```
double price;      // Declaration without initialization
price = 19.99;     // First assignment (initialization)
price = 24.50;     // Updating the value of 'price'
```

However, using a variable without initializing it will cause a compile-time error.

Example of a compile-time error:

```
public class Main {
    public static void main(String[] args) {
        double price;
        // Declaration without initialization

        System.out.println(price);
        /* Error: variable 'price' might not have been
           initialized */

        price = 19.99;
        // First assignment (initialization)
        price = 24.50; // Updating the value of 'price'
        System.out.println(price);
    }
}
```

Output

error: variable price might not have been initialized

Explanation:

1. `double price;`

- declares the variable `price`, but no value is assigned to it yet.

2. `System.out.println(price);`

- **Compile-time Error:** you try to use `price` before initializing it. Java doesn't allow the use of uninitialized local variables, as it could lead to unpredictable behavior.

Correct Initialization

After `price = 19.99;`, the variable is initialized, and you can now safely use or update `price` without causing an error.

To avoid this, always initialize variables before using them in any expressions or print statements.

Example with no compile -time error

```
public class Main {  
    public static void main(String[] args) {  
        double price;  
        // Declaration without initialization  
        price = 19.99;  
        // First assignment (initialization)  
        System.out.println(price);  
        price = 24.50; // Updating the value of 'price'  
        System.out.println(price);  
    }  
}
```

Output:

```
19.99  
24.5
```

Reference Types and Null Values

For reference types (e.g., objects), a variable can be assigned a new object or the special value **null**. The literal **null** is used to indicate that the reference is not pointing to any object in memory. More information on reference types and null values is provided in subsequent lessons.

Using the Scanner Class for Input

Java provides the `Scanner` class to capture user input, allowing the program to accept values from the keyboard during execution.

Steps to use the `Scanner` class:

1. Import the `Scanner` class:

```
import java.util.Scanner;
```

2. Create a `Scanner` object to read input:

```
Scanner input = new Scanner(System.in);
```

3. Use appropriate methods to capture the type of input you need:

- `nextInt()` for integers.
- `nextDouble()` for floating-point numbers.
- `nextLine()` for strings.

Example:

```
import java.util.Scanner;

public class UserInputExample {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.print("Enter your age: ");
        int age = input.nextInt();
        // Reads an integer input

        System.out.print("Enter your name: ");
        String name = input.next();
        // Reads a string input

        System.out.println("Hello");
        System.out.print("Name: ");
        System.out.println(name);
    }
}
```

```

        System.out.print("Age: ");
        System.out.println(age);
    }
}

```

In this example, user input is read and stored in the age and name variables, which are then used in the output.

Explanation:

1. `import java.util.Scanner;`

- This statement imports the Scanner class from Java's java.util package. The Scanner class is used to capture user input from the console.

2. `public class UserInputExample { ... }`

- This defines the main class named UserInputExample. In Java, every program starts with a class declaration, and the main method is where the execution begins.

3. `public static void main(String[] args) { ... }`

- This is the entry point of the program. The main method is executed when you run the program. It's where the logic of your code resides.

4. `Scanner input = new Scanner(System.in);`

- A Scanner object named input is created, which allows the program to capture user input from the console (keyboard).
- System.in refers to the standard input stream, which is typically the keyboard. The Scanner object will use this stream to read data.

5. `System.out.print("Enter your age: ");`

- This prompts the user to enter their age. System.out.print outputs the text "Enter your age: " without moving to a new line, so the user can input their response on the same line.

6. `int age = input.nextInt();`

- This line captures the user's input as an integer using the nextInt() method from the Scanner class. The entered integer is stored in the age variable.
- If the user inputs a value that is not an integer, the program will throw an InputMismatchException.

7. `System.out.print("Enter your name: ");`

- This prompts the user to enter their name. Similar to the earlier prompt, it asks for input without moving to a new line.

8. `String name = input.next();`

- This line captures the user's input as a string using the `next()` method. The entered string is stored in the `name` variable.
- The `next()` method reads a single word (input until a space). If the user enters multiple words, only the first word will be stored.

9. `System.out.println("Hello");`

- This prints the word "Hello" followed by moving to the next line (because of `println`).

10. `System.out.print("Name: ");`

- This prints "Name: " without moving to the next line.

11. `System.out.println(name);`

- This prints the value of the `name` variable (which contains the user's input) and moves to the next line.

12. `System.out.print("Age: ");`

- This prints "Age: " without moving to the next line.

13. `System.out.println(age);`

- This prints the value of the `age` variable and moves to the next line.

Example Interaction:

When the user runs this program, the interaction would look like this:

```
Enter your age: 25
Enter your name: John
Hello
Name: John
Age: 25
```

Input Validation

It is crucial to validate user input to ensure the program works correctly. If a user enters an unexpected value (e.g., a double when an integer is required), the program will throw an `InputMismatchException`. This can be handled with a try-catch block to prevent the program from crashing.

For example:

```
try {
    int number = input.nextInt();
} catch (InputMismatchException e) {
    System.out.println("Invalid input!
                        Please enter a valid number.");
}
```

Let's consider the following program:

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class CorrectInputExample {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        try {
            System.out.print("Enter a number: ");
            int number = input.nextInt();
            // Expecting an integer input

            System.out.print("You entered: ");
            System.out.println(number);
        } catch (InputMismatchException e) {
            System.out.println("Invalid input!
                                Please enter a valid
                                number.");
        }
    }
}
```

Explanation:

1. `import java.util.InputMismatchException;`
 - This imports the `InputMismatchException` class from the `java.util` package. This exception is thrown when a user inputs a value that doesn't match the expected data type, such as entering text when an integer is expected.
2. `import java.util.Scanner;`
 - This imports the `Scanner` class from the `java.util` package. The `Scanner` class is used to read user input from the console.
3. `public class CorrectExample { ... }`
 - This defines the class named `CorrectInputExample`. In Java, every program is defined inside a class.
4. `public static void main(String[] args) { ... }`
 - The `main` method is the entry point of the program, where execution begins.
5. `Scanner input = new Scanner(System.in);`
 - This creates a new `Scanner` object named `input`, which will read input from the console (keyboard). `System.in` refers to the standard input stream.
6. `try { ... }`
 - A `try` block is used to enclose code that might throw an exception. In this case, the code inside the `try` block expects to read an integer input from the user. If the user provides invalid input (e.g., text instead of a number), an exception will be thrown.
7. `System.out.print("Enter a number: ");`
 - This prompts the user to enter a number. It prints the message `"Enter a number: "` on the console without moving to the next line.
8. `int number = input.nextInt();`
 - The `nextInt()` method reads the next integer input from the user. If the user enters a valid integer, it is assigned to the `number` variable. However, if the user enters something that is not an integer (e.g., a string like "hello"), the program will throw an `InputMismatchException`.
9. `System.out.print("You entered: ");`
 - If the user enters a valid integer, this line prints the message `"You entered: "`.
10. `System.out.println(number);`
 - This line prints the value of the `number` variable and moves the cursor to a new line.

11. `catch (InputMismatchException e) { ... }`

- If the user provides invalid input (e.g., a floating-point number), the `InputMismatchException` is caught by the `catch` block. This prevents the program from crashing and allows the program to handle the error gracefully.

12. `System.out.println("Invalid input! Please enter a valid number.");`

- If an `InputMismatchException` is caught, this line prints an error message informing the user that the input was invalid, and they need to enter a valid number.

Example of correct input interaction:

```
Enter a number: 25
You entered: 25
```

The user enters the integer 25, which is read by `input.nextInt()` and stored in the `number` variable. The program then prints "You entered: 25".

Example of Incorrect input (non-integer) interaction:

```
Enter a number: hello
Invalid input! Please enter a valid number.
```

The user enters "hello", which is not an integer. This causes an `InputMismatchException` to be thrown, and the program moves to the `catch` block. The error message "Invalid input! Please enter a valid number." is printed, letting the user know their input was invalid.

The above code is an example of how to handle user input and catch exceptions when the input doesn't match the expected format. The `try` block is used to attempt to read a valid integer from the user, and if the input is invalid, the `catch` block provides a way to manage the error without crashing the program. This helps ensure that the program runs smoothly even when the user makes a mistake.

Activities

1. What is the purpose of an assignment statement in Java, and how does it differ from a comparison statement?
2. Why must a variable be initialized before it is used in an expression?
3. How can Java programs get input from the user?
4. What is an `InputMismatchException`, and how can it be prevented?
5. How can reference types in Java be assigned values, and what is the significance of null?
6. Write a Java program that prompts the user to input two numbers.

The program should calculate and display the sum, difference, product, and quotient of the two numbers.
7. Write a Java program that prompts the user to input a temperature in Celsius, and the program converts it to Fahrenheit using the formula $F = (C * 9/5) + 32$.
8. Write a Java program that prompts the user to input a principal amount, rate of interest, and time in years.

Calculate and print the simple interest using the formula $\text{Interest} = (\text{Principal} * \text{Rate} * \text{Time}) / 100$.

9. Identify the error in the following Java code and correct the code to ensure it runs properly:

```
import java.util.Scanner;

public class Test {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        System.out.println("Enter your age: ");
        int age = input.nextInt();

        System.out.println("Enter the price: ");
        double price;
        System.out.println(price);

        price = input.nextDouble();
    }
}
```

Key Takeaways

- Every variable must be assigned a value before it can be used in an expression. Variables must be initialized first with a value from a compatible data type.
- Reference types can be assigned a new object or the special `null` value.
- Assignment statements store the result of an evaluated expression into a variable.
- The `Scanner` class allows for user input, making programs interactive.
- Input validation ensures the program can handle unexpected data from users effectively.

Key Terms

assignment statement

A statement that assigns a value to a variable.

expression

A combination of variables, literals, and operators that is evaluated to produce a value.

initialization

The first time a variable is assigned a value.

null

A special value that indicates a reference variable is not associated with any object.

Scanner class

A Java class used for receiving user input from the console.

LESSON 5

Casting and Range of Variables

AP Learning Objectives

1.5.A Develop code to cast primitive values to different primitive types in arithmetic expressions and determine the value that is produced as a result.

1.5.B Describe conditions when an integer expression evaluates to a value out of range.

1.5.C Describe conditions that limit accuracy of expressions.

In this lesson, we explore **casting** and the **range** of primitive variables in Java. Casting allows us to convert one data type into another, which is useful when working with arithmetic operations that involve different data types. Understanding how casting works and the range limitations of primitive types helps avoid errors in your programs. By using explicit casts and considering issues like integer overflow and floating-point precision, you can write more reliable code.

Casting Primitive Values

Casting refers to converting one primitive data type into another. Java provides two types of casting: implicit (widening) and explicit (narrowing).

- **Implicit Casting** (Widening): This happens automatically when converting a smaller data type to a larger one (e.g., `int` to `double`). Since no data is lost, you don't need to explicitly cast the value.
- **Explicit Casting** (Narrowing): This is required when converting a larger data type to a smaller one (e.g., `double` to `int`), as it may result in data loss. To explicitly cast, place the target data type in parentheses before the value or variable.

Casting Between int and double

Two scenarios occur when casting between `int` and `double` values.

Casting from `double` to `int`:

When a `double` is cast to an `int`, the fractional part is truncated (it leaves off all digits after the decimal).

Example:

```
double decimalValue = 9.8;
int integerValue = (int) decimalValue;
// Result: integerValue is 9
```

Widening from `int` to `double`:

No cast is required because Java automatically converts `int` to `double` when needed.

Example:

```
int wholeNumber = 5;
double result = wholeNumber + 2.3;
// Result: result is 7.3
```

Rounding `double` Values

When casting a `double` to an `int`, Java truncates the decimal part. If rounding is needed, you can add or subtract 0.5 before casting to achieve rounding.

Example of Truncation:

```
double value = 9.99;
int truncated = (int) value; // Truncates the decimal
                             part
System.out.println(truncated); // Output: 9
```

Example of Rounding:

```
double value = 9.99;
int rounded = (int)(value + 0.5); // Adds 0.5 to round up
```

Integer Overflow and Range of Variables

Each primitive data type in Java has a specific **range** based on the memory allocated to it. For instance, an `int` is a 32-bit signed integer and can store values from **-2,147,483,648 to 2,147,483,647**.

Constants for `int` Range

Java provides constants to represent the maximum and minimum values for `int`:

- `Integer.MAX_VALUE`: 2,147,483,647
- `Integer.MIN_VALUE`: -2,147,483,648

Integer Overflow

When an arithmetic expression results in a value outside the range of an `int`, integer **overflow** occurs. Rather than throwing an error, Java wraps the value around to stay within the allowed range. This can lead to unexpected results.

Example of overflow:

```
int largeValue = Integer.MAX_VALUE;
int overflowedValue = largeValue + 1;
// Result: overflowedValue is -2,147,483,648
```

The opposite situation is called integer underflow.

Note that in both cases, when you exceed the range of an `int`, Java does not throw an error. Instead, the value wraps around from the maximum to the minimum (and vice versa), which can lead to unexpected behavior.

Limiting Accuracy with `double`

While `double` values offer more precision than `int`, they are still limited by the memory allocated (64 bits). This limitation can cause rounding errors in large or precise calculations, where some accuracy is lost due to the inability to represent all decimal places.

Example of rounding error:

```
double result = 0.7 + 0.1;
System.out.println(result); // Output: 0.7999999999999999
```

Double Precision and Rounding Errors

Unlike integers, floating-point numbers (such as `double`) have limited **precision**, leading to rounding errors. This occurs because the memory allocated for `double` cannot store an infinite number of decimal places.

To avoid rounding errors, it's best to use `int` for calculations where exact results are needed, like counting or indexing.

For Review Purpose Only

Activities

1. Explain the difference between implicit and explicit casting in Java.
2. How do `double` types use the computer's resources for more arithmetic precision?
3. What is the default way that Java handles integer overflow?
4. Write a Java program that casts a `double` to an `int` and prints both the truncated and rounded versions of the number.

Input:

7.56

Output:

Truncated: 7
Rounded: 8

5. Write a program that demonstrates integer overflow by adding 1 to `Integer.MAX_VALUE` and subtracting 1 from `Integer.MIN_VALUE`.

Observe and explain the results.

6. What is the output of the following code and explain the difference between the two outputted values:

```
public class Test {  
    public static void main(String[] args) {  
        // Implicit Casting (Widening)  
        int smallNumber = 42;  
        double largeNumber = smallNumber;  
  
        // Explicit Casting (Narrowing)  
        double decimalValue = 9.7;  
        int roundedValue = (int) decimalValue;  
    }  
}
```

```
        // Printing values
        System.out.println("Implicit Casting: " +
        largeNumber);
        System.out.println("Explicit Casting: " +
        roundedValue);
    }
}
```

Key Takeaways

- `int` values are limited to the range $-2,147,483,648$ to $2,147,483,647$.
- When an arithmetic operation results in a value beyond the allowed range for `int`, it causes the value to "wrap around."
- `double` values can introduce round-off errors when precision exceeds available memory.

Key Terms

casting

Temporarily converting one data type into another using the cast operator.

range of int

`int` values are limited to the range $-2,147,483,648$ to $2,147,483,647$.

overflow

When an arithmetic operation results in a value beyond the allowed range for `int`, causing the value to "wrap around."

floating-point precision

`double` values can introduce round-off errors when precision exceeds available memory.

LESSON 6

Compound Assignment Operators

AP Learning Objectives

1.6.A Develop code for assignment statements with compound assignment operators and determine the value that is stored in the variable as a result.

Compound Assignment Operators

In Java, **compound assignment operators** simplify common operations by combining arithmetic operations with assignment. These operators allow you to perform an operation on a variable and assign the result back to the same variable in one step.

Examples of Compound Assignment Operators:

+=:

Adds the right-hand operand to the left-hand operand and assigns the result to the left-hand operand.

```
int x = 5;  
x += 3; // Equivalent to: x = x + 3; x is now 8
```

-=:

Subtracts the right-hand operand from the left-hand operand.

```
int y = 10;  
y -= 2; // Equivalent to: y = y - 2; y is now 8
```

Other compound assignment operators include ***=**, **/=**, and **%=** for multiplication, division, and modulo operations.

Increment and Decrement Operators

Java also offers **increment** (`++`) and **decrement** (`--`) operators to increase or decrease the value of a variable by 1. These operators can be used in two forms:

- **Prefix** (`++x` or `--x`): The variable is modified **before** the expression is evaluated.
- **Postfix** (`x++` or `x--`): The expression is evaluated **before** the variable is modified.

Prefix increment example:

```
int x = 5;
int y = ++x; // Prefix: x is incremented to 6, then y is
              assigned 6
System.out.print("x: ");
System.out.println(x);
System.out.print("y: ");
System.out.println(y);
```

Step-by-Step Execution:

1. `int x = 5;`
 - The variable `x` is initialized with the value 5.
2. `int y = ++x;`
 - **Prefix increment** (`++x`): `x` is incremented **before** its value is assigned to `y`.
 - `x` becomes 6, and then `y` is assigned the value of `x`, so `y = 6`.
3. `System.out.print("x: ");`
 - Prints the string `"x: "` without moving to a new line.
4. `System.out.println(x);`
 - Prints the value of `x`, which is 6 and moves the cursor to the next line.
5. `System.out.print("y: ");`
 - Prints the string `"y: "` without moving to a new line.
6. `System.out.println(y);`
 - Prints the value of `y`, which is 6 and moves the cursor to the next line.

Output:

```
x: 6
y: 6
```

Postfix increment example:

```
int a = 5;
int b = a++; /* Postfix: b is assigned the current value
of a (5), then a is incremented to 6 */
System.out.print("a: ");
System.out.println(a);
System.out.print("b: ");
System.out.println(b);
```

Step-by-Step Execution:

1. `int a = 5;`
 - The variable `a` is initialized with the value 5.
2. `int b = a++;`
 - **Postfix increment** (`a++`): The value of `a` is assigned to `b` before `a` is incremented.
 - `b` is assigned the current value of `a`, which is 5.
 - Then, `a` is incremented by 1, so `a` becomes 6.
3. `System.out.print("a: ");`
 - Prints the string `"a: "` without moving to a new line.
4. `System.out.println(a);`
 - Prints the current value of `a`, which is 6, and moves to a new line.
5. `System.out.print("b: ");`
 - Prints the string `"b: "` without moving to a new line.
6. `System.out.println(b);`
 - Prints the value of `b`, which is 5, and moves to a new line.

Output:

```
a: 6
b: 5
```

Prefix decrement example:

```
int x = 5;
int y = --x; // Prefix: x is decremented to 4, then y is
              assigned 4
System.out.print("x: ");
System.out.println(x);
System.out.print("y: ");
System.out.println(y);
```

Step-by-Step Execution:

1. `int x = 5;`
 - The variable `x` is initialized with the value 5.
2. `int y = --x;`
 - **Prefix decrement** (`--x`): `x` is decremented **before** its value is assigned to `y`.
 - `x` is reduced by 1, so `x` becomes 4.
 - Then, the value of `x` (which is now 4) is assigned to `y`, so `y = 4`.
3. `System.out.print("x: ");`
 - Prints the string `"x: "` without moving to a new line.
4. `System.out.println(x);`
 - Prints the current value of `x`, which is 4, and moves the cursor to the next line.
5. `System.out.print("y: ");`
 - Prints the string `"y: "` without moving to a new line.
6. `System.out.println(y);`
 - Prints the value of `y`, which is 4, and moves to the next line.

Output:

```
x: 4
y: 4
```

Postfix decrement example:

```
int a = 5;
int b = a--; /* Postfix: b is assigned the current value
of a (5), then a is decremented to 4 */
System.out.print("a: ");
System.out.println(a);
System.out.print("b: ");
System.out.println(b);
```

Step-by-Step Execution:

1. `int a = 5;`
 - The variable `a` is initialized with the value 5.
2. `int b = a--;`
 - **Postfix decrement (`a--`):** The current value of `a` is assigned to `b`, before `a` is decremented.
 - `b` is assigned the current value of `a`, which is 5.
 - Then, `a` is decremented by 1, so `a` becomes 4.
3. `System.out.print("a: ");`
 - Prints the string `"a: "` without moving to a new line.
4. `System.out.println(a);`
 - Prints the current value of `a`, which is 4, and moves the cursor to the next line.
5. `System.out.print("b: ");`
 - Prints the string `"b: "` without moving to a new line.
6. `System.out.println(b);`
 - Prints the value of `b`, which is 5, and moves the cursor to the next line.

Output:

```
a: 4
b: 5
```

Activities

1. Explain the purpose of compound assignment operators in Java and how they simplify arithmetic operations.
2. How does the `%=` (modulus assignment) operator work, and in give an example of a situation where it can be used.
3. Write a Java program that calculates and prints the result of the following operations:
 1. The sum of two integers: `15 + 27`
 2. The product of two integers: `7 * 8`
 3. The result of dividing two integers: `100 / 9` (observe how integer division works)
 4. The remainder when dividing 100 by 9 (using the modulo operator `%`)

Example Output:

```
Sum: 42
Product: 56
Quotient: 11
Remainder: 1
```

4. Write a program that declares an integer `x` with an initial value of 10.

Use compound assignment operators to:

1. Add 5 to `x`.
2. Multiply `x` by 3.
3. Subtract 7 from `x`.
4. Print the final value of `x`.

Example Output:

```
Final value of x: 38
```

5. Modify the previous activity to use parentheses and ensure that operations occur in the order you intend.

For example, calculate $(15 + 27) * 2$ instead of $15 + 27 * 2$.

These tasks will help you practice arithmetic operations, string formatting, and output generation, while improving your understanding of operator precedence and debugging.

For Review Purpose Only

Key Takeaways

- Compound assignment operators simplify common operations like addition and subtraction.
- Increment (++) and decrement (--) operators are shorthand for adding or subtracting 1 from a variable.
- **Prefixes and postfixes** modify the variable before the expression is evaluated.

Key Terms

compound assignment operators

Shortcuts for performing an operation and assignment in a single step. Examples include:

- += (Addition assignment)
- -= (Subtraction assignment)
- *= (Multiplication assignment)
- /= (Division assignment)
- %= (Modulo assignment)

increment operator (++)

Increases a variable's value by 1.

decrement operator (--)

Decreases a variable's value by 1.

prefix form

Modifies the value before the expression is evaluated (++x or --x).

postfix form

Evaluates the expression before modifying the value (x++ or x--).

LESSON 7

Application Program Interface (API) and Libraries

AP Learning Objectives

1.7.A Identify the attributes and behaviors of a class found in the libraries contained in an API.

In this lesson, we will explore **Application Program Interfaces (APIs)** and **libraries** in Java. APIs and libraries are crucial for allowing developers to use prewritten code to solve common problems quickly. This makes coding simpler, as you don't need to write everything from scratch. By using these tools, your programs become more efficient and easier to maintain.

Understanding APIs

An **API (Application Programming Interface)** is a set of tools that allows different parts of a program, or even different programs, to communicate with each other. In Java, APIs are available as libraries that help developers perform tasks like taking input from users or displaying output.

Simple Example of Using an API:

```
import java.util.Scanner;

public class APISimpleExample {
    public static void main(String[] args) {
        // Create a Scanner object to get user input
        Scanner input = new Scanner(System.in);

        // Ask the user to enter a number
        System.out.print("Enter a number: ");

        int number = input.nextInt(); // Reads an integer
                                     // input
    }
}
```

```

        // Display the number entered by the user
        System.out.println(number);
    }
}

```

Output:

```

Enter a number: 2
2

```

In this example, we use the `Scanner` class from the **Java API**, which is part of the `java.util` package. The `Scanner` class allows us to easily read user input, making interaction with the user simple. Without this API, handling user input would be much more complicated.

Using Java Libraries

A **library** in Java is a collection of classes and methods that developers can use to simplify programming tasks. The Java Standard Library provides a wide range of functionality, such as reading input, generating random numbers, and manipulating data. Libraries prevent you from needing to write code for common tasks yourself, reducing errors and saving time.

Example Using the Random Library:

```

import java.util.Random;

public class RandomExample {
    public static void main(String[] args) {
        // Create a Random object
        Random random = new Random();

        // Generate a random number between 0 and 9
        int randomNumber = random.nextInt(10);

        // Display the random number
        System.out.println(randomNumber);
    }
}

```

Output:

7

In this example, we use the `Random` class from the `java.util` package. The `Random` class provides a way to generate random numbers, which can be useful in games, simulations, or any program where random values are needed.

API Documentation

To use APIs effectively, it's important to understand the methods and classes available to you.

API documentation explains how to use these tools, what input they require, and what output they provide. Java's official documentation offers a detailed reference for every class and method in the standard library, making it easier for developers to understand how to use the provided tools.

For example, Java API documentation for the `Scanner` class describes the methods you can use to read different types of input (like `nextInt()` for integers or `nextLine()` for strings).

Activities

1. Explain what APIs are and why they are useful in software development.
2. What is the difference between APIs and libraries in Java?
3. Write a Java program that asks the user for five random number and the calculates the mean of those inputted numbers.
4. Write a simple Java program that:
 1. Uses the Scanner class to ask the user to enter two numbers.
 2. Adds the two numbers together and prints the result.

Key Takeaways

- **APIs** offer predefined methods and classes that make coding easier by providing solutions for common tasks.
- **Libraries** provide reusable code that reduces the complexity of your programs. In this unit, we used the Scanner and Random libraries to handle user input and generate random numbers.
- **API Documentation** helps you understand how to use the methods and classes available in a library. It describes each method's purpose, required input, and expected output.

Key Terms

API (Application Programming Interface)

A set of tools and protocols that allows programs to interact with each other.

library

A collection of prewritten classes and methods that can be used in your programs to simplify common tasks.

Java API documentation

A detailed reference for using Java's built-in classes and methods, helping developers understand how to use them properly.

LESSON 8

Documentation with Comments

AP Learning Objectives

1.8.A Describe the functionality and use of code through comments.

In Java programming, comments play a crucial role in making code readable, maintainable, and understandable. While comments are not executed by the compiler, they provide explanations of what the code is doing, helping others (or even your future self) to understand the logic. Well-documented code ensures that complex logic, assumptions, and objectives are clear to anyone who works with the code.

Type of Comment in Java

By using effective commenting, you can make your code easier to read, debug, and maintain, improving collaboration and ensuring that your programs remain understandable even as they grow in complexity.

Java provides three types of comments to document code:

Single-line comments (`//`): Used for brief explanations that apply to a single line or statement.

Multi-line comments (`/* ... */`): Useful for longer descriptions that span several lines of code or provide an overview of a section.

Javadoc comments (`/** ... */`): Used for generating API documentation and are typically placed above classes, methods, and fields to describe their purpose, parameters, and return values.

Benefits of Comments:

- Comments help explain why certain decisions were made in the code, not just what the code does.
- They enhance the readability and maintainability of code by breaking down complex logic into understandable steps.

- Documenting preconditions (requirements before a method is executed) and postconditions (guarantees after execution) ensures that a method's contract is clear and can help prevent misuse.

Examples of Comments:

Single-line Comments (//):

Use these for short explanations or to describe specific lines of code.

```
int age = 25; // Initialize age with 25
age += 1; // Increment age by 1
```

Multi-line Comments (/* ... */):

Ideal for more detailed explanations or when commenting on multiple lines of code at once.

```
/*
 * This block of code calculates the total price
 * by multiplying the quantity of items by their price.
 * It then applies a 10% discount if the total is above
 * $100.
 */
double price = 50.0;
int quantity = 3;
double total = price * quantity;
if (total > 100) {
    total *= 0.9; // Apply a 10% discount
}
```

Javadoc Comments (/** ... */):

These comments are used for documenting classes, methods, and fields, and they can be processed by the javadoc tool to generate HTML documentation for your code. Javadoc comments should explain the purpose of the method or class, describe parameters, and indicate return values.

```

/**
 * This method calculates the product of two integers.
 * @param a the first integer
 * @param b the second integer
 * @return the product of a and b
 */
public int multiply(int a, int b) {
    return a * b;
}

```

Preconditions and Postconditions:

A precondition is a requirement that must be true just before a method is executed in order for it to function correctly. Methods are not expected to check whether their preconditions are met — it's the caller's responsibility. A postcondition, on the other hand, is a condition that must be true after the method has finished running. Postconditions describe the expected outcome of the method, such as the return value or changes to an object's attributes.

```

/**
 * Divides two numbers.
 *
 * @author Binary Logic
 * @version 0.1
 *
 * Preconditions:
 * - The divisor (b) must not be zero.
 *
 * Postconditions:
 * - Returns the result of dividing a by b.
 *
 * @param a the dividend
 * @param b the divisor
 * @return the quotient of a divided by b
 * @throws ArithmeticException if b is zero
 */

```

```
public double divide(int a, int b) throws
ArithmeticException {
    if (b == 0) {
        throw new ArithmeticException("Divisor cannot be
zero.");
    }
    return (double) a / b;
}
```

For Review Purpose Only

Activities

1. Explain what Javadoc comments are and how they are used.
2. Analyze the reason why multiple types of comments are used in Java software development.
3. Write a Java program that:
 1. Asks the user to enter a number.
 2. Checks whether the number is even or odd and prints 0 for even and 1 for odd.
 3. Use single-line comments to explain variable declarations.
 4. Use multi-line comments to describe the logic of the program.
 5. Use Javadoc comments to explain the method.

Key Takeaways

- **Comments** are a powerful tool for explaining code, making it easier to understand and maintain.
- **Single-line comments** are ideal for short explanations, while multi-line comments are useful for describing larger blocks of code.
- **Javadoc comments** are specifically used to generate documentation for classes and methods, and they should clearly explain the method's purpose, inputs, and outputs.
- **Preconditions and postconditions** ensure that a method's requirements and guarantees are clearly defined.

Key Terms

single-line comment (//)

A comment used to describe a specific line of code.

multi-line comment (/* ... */)

A comment used to describe multiple lines of code or provide an overview of a section.

Javadoc comment (/** ... */)

A comment used to generate API documentation and describe methods, parameters, and return values.

precondition

A condition that must be met before a method executes.

postcondition

A condition that is guaranteed after a method executes.

LESSON 9

Method Signatures

AP Learning Objectives

1.9.A Identify the correct method to call based on documentation and method signatures.

1.9.B Describe how to call methods.

In Java, **methods** are fundamental building blocks that allow us to group code into reusable and manageable sections. A method is a named block of code that only runs when called, and procedural abstraction enables programmers to use methods without knowing the internal implementation. This lesson covers method signatures, parameters, how to call methods, and the concepts of void and non-void methods.

Methods and Method Signatures

A **method** is a named block of code enclosed in braces `{ }`. A **method signature** defines how a method is declared and includes:

- **Method name:** The name that is used to call the method.
- **Parameter list:** A list of types and variables that the method accepts. A method signature without parameters consists of just the method name and empty parentheses.

Consider for example a method named “greet” with no parameter as follows:

```
public static void greet() { // Method signature: greet()
    System.out.println("Hello!");
}
```

Method Signature:

- **public**: This is an access modifier, meaning that the method is accessible from any other part of the program or other classes. It defines the visibility of the method.
- **static**: This means the method belongs to the class itself, not to any specific object created from the class. You can call a static method without creating an instance of the class. We'll explore this keyword in more detail in the next lesson.
- **void**: This means the method does not return a value — it simply performs an action and then finishes. For example, printing a message to the screen is an action, but there's nothing being "sent back" after it's done. So far, every method you've seen has been void, but soon you'll learn that methods can also return values — like a number from a calculation — which can be stored in a variable or used in other parts of the program.
- **greet**: This is the method name, which is used to call or invoke the method.
- **()**: The empty parentheses indicate that the method does not accept any parameters. It will always perform the same action when called, regardless of any outside input.

Method Body:

```
| System.out.println("Hello!");
```

- This line is the actual action that the method performs. It uses the `System.out.println()` function to print the text "Hello!" to the console.
- The `System.out.println()` function prints the message and moves the cursor to the next line, meaning any subsequent output will be printed on the line below "Hello!".

How the Method Works:

- When the `greet` method is called, it performs a single action: printing "Hello!" to the console.
- Since this method doesn't accept any parameters, its behavior will always be the same, regardless of how many times or from where it's called.

Example of calling the method:

```
| public static void main(String[] args) {  
|     greet();  
| }
```

When this line is executed, the `greet()` method runs, and the console will display:

```
Hello!
```

Procedural Abstraction

This allows a programmer to use a method by understanding what it does, without needing to know its internal workings. When calling a method like `greet()`, a programmer only needs to know that it prints a greeting, but not how it does so.

Parameters

A **parameter** is a variable defined in the method header and used inside the method body. When a method is called, **arguments** (values) are passed to these parameters. The number and type of arguments must match the method's parameter list.

Consider for example a method named “`greetUser`” with one parameter `int userAge` as follows:

```
public void greetUser(int userAge) {  
    System.out.println("Hello!");  
    System.out.print("Your age is: ");  
    System.out.println(userAge);  
}
```

Method Signature:

- **public**: This is an **access modifier**. It means the method is accessible from anywhere in the program or even other classes.
- **void**: This indicates that the method does not return any value. The purpose of this method is to perform an action (printing messages) without returning any result.
- **greetUser**: This is the **method name**. It is used to call or invoke the method from other parts of the code.
- **int userAge**: This is the **parameter** for the method. The method accepts an integer (`int`) value when it is called. This value is passed in as the argument and is referred to as `userAge` within the method.

Method Body:

```
| System.out.println("Hello!");
```

This line prints the message "Hello!" to the console. It uses `System.out.println()`, which moves the cursor to the next line after printing.

```
| System.out.print("Your age is: ");
```

This prints the message "Your age is: " but does not move the cursor to the next line. The `System.out.print()` method prints the message, keeping the cursor on the same line, so the next output will appear immediately after this message.

```
| System.out.println(userAge);
```

This prints the value of `userAge`, which is the integer passed to the method when it's called. After printing the value, `System.out.println()` moves the cursor to the next line.

How the Method Works:

Let's assume we call the method like this:

```
| greetUser(25);
```

1. The method prints "Hello!" on one line.
2. Then it prints "Your age is: " and keeps the cursor on the same line.
3. Finally, it prints 25, which is the value of `userAge`, and moves the cursor to a new line.

Example Output:

```
Hello!  
Your age is: 25
```

This method is a simple example of how arguments can be passed to methods in Java, and how you can use them to customize the output.

Calling Methods

When a method is called, the program pauses its current execution and jumps to the method. After the method finishes execution, the program resumes from where the method was called.

Calling Void Methods

A **void method** does not return any value and is called without being part of an expression.

Example:

```
public void displayMessage() {
    System.out.println("This is a void method.");
}

// Calling the method
displayMessage();
// Simply calls the method; no value is returned.
```

Calling Non-Void Methods

A **non-void method** returns a value. To use the returned value, the method call must either be part of an expression or the result stored in a variable.

Example:

```
public class Main {
    public static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int result = add(10, 5);
        System.out.println(result);
    }
}
```

Or alternatively,

```
public class Main {  
    public static int add(int a, int b) {  
        int result = a + b;  
        return result;  
    }  
  
    public static void main(String[] args) {  
        int sum = add(10, 5);  
        System.out.println(sum);  
    }  
}
```

15

Method Signature:

- **public**: This is the **access modifier**, meaning the method can be accessed from any part of the program or other classes.
- **int**: This is the **return type**, indicating that the method will return an integer value.
- **add**: This is the **method name**, which is used to call the method.
- **(int a, int b)**: These are the **parameters** that the method accepts. It takes two integers, a and b, which will be provided when the method is called.

Method Body:

```
int result = a + b;
```

- This line adds the two integer parameters **a** and **b** and stores the sum in a variable named **result**.

```
return result;
```

- This returns the value stored in **result** to the part of the program that called the method. Since the return type is **int**, the method must return an integer.

Method Call:

```
| int sum = add(10, 5);
```

- Here, the `add` method is called with arguments `10` and `5`. These values are passed to the parameters `a` and `b`, and the method returns the sum `15`.
- The result of the method (`15`) is stored in the variable `sum`.

Output:

```
| System.out.println(sum);
```

- The value stored in `sum` (which is `15`) is printed to the console.

Method Overloading

Overloading occurs when multiple methods have the same name but different parameter lists (signatures). This allows methods to handle different types of inputs or different numbers of arguments.

Example:

```
| public void printNumber(int number) {  
|     System.out.println(number);  
| }  
  
| public void printNumber(double number) {  
|     System.out.println(number);  
| }
```

In this example, the method `printNumber` is overloaded to handle both `int` and `double` values.

Example with Overloaded Methods (Same Method Name, Different Argument Types):

```
public class MathOperations {  
    // Method to multiply two integers  
    public static int multiply(int a, int b) {  
        return a * b;  
    }  
  
    // Overloaded method to multiply three integers  
    public static int multiply(int a, int b, int c) {  
        return a * b * c;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(multiply(3, 4));  
        // Passing two arguments  
  
        System.out.println(multiply(2, 3, 4));  
        //Passing three arguments  
    }  
}
```

Output:

```
12  
24
```

Explanation:

- The method `multiply` is **overloaded**—there are two versions of the method, one that takes two `int` arguments and one that takes three `int` arguments.
- The call `multiply(3, 4)` passes two arguments, while `multiply(2, 3, 4)` passes three arguments.

Example with Mismatched Arguments (Will Cause an Error):

```
public class Main {  
    public static void greetUser(int userAge) {  
        System.out.println("Hello!");  
        System.out.print("Your age is: ");  
        System.out.println(userAge);  
    }  
  
    public static void main(String[] args) {  
        greetUser(34.5); // ERROR: 34.5 is not an int  
    }  
}
```

Explanation:

- The method `greetUser` expects an `int` argument, but the method call `greetUser(34.5)` passes a `double`.
- This will cause a **compile-time error** because the argument type does not match the expected parameter type.

Flow of Control

A method call interrupts the sequential flow of a program. When a method is called, the program executes the method's statements, and once finished, control is returned to the point just after the method was called.

Example: Method Call Interrupting the Flow

```
public class MethodFlowExample {  
    public static void main(String[] args) {  
        System.out.println("Program starts...");  
  
        // Calling the first method  
        printMessage(); /* Program jumps to execute  
                        printMessage() */  
    }  
}
```

```

        System.out.println("Back in main method.");

        // Calling the second method
        int result = multiply(5, 3);

        // Program jumps to execute multiply()
        System.out.print("Result of multiplication: ");
        System.out.println(result);

        System.out.println("Program ends.");
    }

    // A simple method that prints a message
    public static void printMessage() {
        System.out.println("Inside the printMessage
method.");
    }

    // A method that multiplies two numbers and returns
the result
    public static int multiply(int a, int b) {
        System.out.println("Inside the multiply method.");
        return a * b;
    }
}

```

Output:

```

Program starts...
Inside the printMessage method.
Back in main method.
Inside the multiply method.
Result of multiplication: 15
Program ends.

```

Explanation:

1. Sequential Flow:

- The program starts with the main method and prints "Program starts...".

2. First Method Call - `printMessage()`:

- The program encounters the `printMessage()` call and **jumps** to the `printMessage` method.
- Inside the `printMessage()` method, it prints "Inside the `printMessage` method."
- After the method finishes, the program **returns** to the `main` method and continues from the line **right after** the method call, printing "Back in `main` method."

3. Second Method Call - `multiply(5, 3)`:

- The program encounters the `multiply(5, 3)` call and **jumps** to the `multiply` method.
- Inside the `multiply()` method, it prints "Inside the `multiply` method.", multiplies 5 and 3, and returns the result (15).
- The result 15 is stored in the variable `result` and the program continues from where it left off, printing "Result of multiplication: 15".

4. Program End:

- After both methods are called and control is returned back to the `main` method, the program prints "Program ends." and completes execution.

Activities

1. Explain how procedural abstraction helps programmers understand large and complex programs.
2. What type of error occurs when we assign the result of a method with a `void` signature to a parameter?
3. Give two examples where you might want to overload a method with multiple instances.
4. Create a Java program with the following:
 1. A void method `printMessage` that prints a welcome message.
 2. A non-void method `multiply` that takes two integers and returns their product.
 3. Overload the `multiply` method to handle three integers.
 4. Call all methods and print their results.
5. Identify the error in the following Java code and correct the code to ensure it runs properly:

```
public class Greeting {  
    public void greetUser(int userAge) {  
        System.out.println("Hello!");  
        System.out.print("Your age is: ");  
        System.out.println(userAge);  
    }  
  
    public static void main(String[] args) {  
        greetUser(25.  
    }
```

6. Identify the error in the following Java code and correct the code to ensure it runs properly:

```
public class SquareCalculator {  
    public void square(int num) {  
        return num * num;  
    }  
  
    public static void main(String[] args) {  
        SquareCalculator calc = new SquareCalculator();  
        int result = calc.square(5);  
        System.out.println("Square: " + result);  
    }  
}
```

7. Identify the error in the following Java code and correct the code to ensure it runs properly:

```
public class Calculator {  
    // Overloaded methods for addition  
    public static int add(int a, int b) {  
        return a + b;  
    }  
  
    public static double add(int a, double b) {  
        return a + b;  
    }  
  
    public int add(int a, int b, int c) {  
        return a + b + c;  
    }  
  
    public static void main(String[] args) {
```

```
        System.out.println("Sum 1: " + add(5, 10));
        System.out.println("Sum 2: " + add(7, 2.5));
        System.out.println("Sum 3: " + Calculator.
add(1, 2, 3));
    }
}
```

For Review Purpose Only

Key Takeaways

- Methods allow for reusable and organized code. They can take parameters to work with different inputs.
- **Void methods** perform actions without returning any value, while **non-void methods** return values that can be used in expressions or stored in variables.
- **Overloading** allows methods with the same name to handle different types or numbers of arguments.
- Method calls interrupt the normal flow of execution and return control to the calling code once the method is finished.

Key Terms

method signature

The method name and parameter list, used to identify a method.

parameters

Variables defined in the method signature that receive values (arguments) when the method is called.

void method

A method that does not return any value.

non-void method

A method that returns a value of a specific type.

arguments

Values passed to a method when it is called.

overloading

Defining multiple methods with the same name but different parameter lists.

flow of control

The order in which statements in a program are executed, which can be altered by method calls.

LESSON 10

Calling Class Methods

AP Learning Objectives

1.10.A Develop code to call class methods and determine the result of those calls.

This lesson discusses how to call class methods in Java and determine the result of those method calls.

Class Methods

In Java, there are two main types of methods: **class methods** and **instance methods**.

Class methods are associated with the class itself and can be called without creating an object. They include the keyword `static` in the method header and typically perform operations that do not depend on the state of any particular object. In contrast, instance methods are tied to specific objects or instances of the class, as we will discuss in a later lesson. These methods must be called on an object of the class and operate on data that belongs to that specific instance.

Class Methods:

Class methods, also called **static methods**, are shared among all instances of the class. They do not depend on an instance of the class and are called directly using the class name followed by the method name.

The Dot Operator for Calling Static Methods

In Java, static methods belong to the class itself rather than to any particular object of that class. To call a static method, you use the dot operator (`.`) along with the class name instead of an object. This is different from instance methods, which require an object to call them.

Syntax for Calling Static Methods:

```
ClassName.methodName();
```

- **ClassName**: The name of the class that contains the static method.
- **methodName**: The name of the static method to be called.

Examples of Calling Static Methods Using the Dot Operator

Example with a Custom Class:

```
public class MathOperations {  
    // Static method to add two numbers  
    public static int add(int a, int b) {  
        return a + b;  
    }  
    // Static method to multiply two numbers  
    public static int multiply(int a, int b) {  
        return a * b;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        // Calling static methods using the dot operator and  
        // class name  
        int sum = MathOperations.add(10, 5);  
        // Calls MathOperations.add()  
        int product = MathOperations.multiply(3, 4);  
        // Calls MathOperations.multiply()  
  
        System.out.println(sum);  
        System.out.println(product);  
    }  
}
```

Output:

```
15  
12
```

In this example:

- `MathOperations.add(10, 5)`: The dot operator is used with the class name `MathOperations` to call the `add` static method.
- `MathOperations.multiply(3, 4)`: Similarly, the dot operator is used to call the `multiply` static method.

The dot operator allows direct access to static methods through the **class name**, as static methods are not tied to any specific instance of the class. This makes static methods convenient for utility or helper functions, such as mathematical calculations or commonly used operations that don't require object-specific data.

Example:

```
public class MathOperations {  
    public static int add(int a, int b) {  
        return a + b;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        int result = MathOperations.add(5, 3);  
        System.out.println(result);  
    }  
}
```

Output:

8

In this example, the `add` method is a static method in the `MathOperations` class. It is called using the class name `MathOperations` without creating an instance of the class.

Calling Class Methods from the Same Class:

When calling class methods within the same class, you can omit the class name and directly call the method.

Example:

```
public class Calculator {  
    public static int multiply(int a, int b) {  
        return a * b;  
    }  
    public static void main(String[] args) {  
        int result = multiply(4, 5); //Direct call without  
class name  
        System.out.println(result);  
    }  
}
```

Output:

20

Another example of a static method that calculates the square of a given number.

```
public class Utility {  
    public static int square(int number) {  
        return number * number;  
    }  
}  
public class Main {  
    public static void main(String[] args) {  
        int result = Utility.square(6);  
        System.out.println(result);  
    }  
}
```

Output:

36

Activities

1. Describe the difference between class methods and instance methods.
2. How are static methods used inside the same class in Java?
3. Write a complete Java program that calculates the cube of a number by using a static method.

The cube of a number is obtained by raising it to the power of three (i.e., multiplying the number by itself twice). Your program should include a static method named `cube` that takes a single `int` or `double` parameter and returns its cube.

4. Write a Java program that asks the user to input the dimensions of two sides of a right triangle and uses the Pythagorean theorem to calculate the third side.
5. Identify the error in the following Java code and correct the code to ensure it runs properly:

```
public class MathOperations {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        int sum = MathOperations.add(10, 5);  
        System.out.
```

6. Identify the errors in the following Java code and correct the code to ensure it runs properly:

```
public class TemperatureConverter {
    public static double toCelsius(double fahrenheit)
    {
        return (fahrenheit - 32) * 5 / 9;
    }

    public double toFahrenheit(double celsius) {
        return (celsius * 9 / 5) + 32;
    }
}

public class Main {
    public static void main(String[] args) {
        double celsius = toCelsius(98.6);
        double fahrenheit = TemperatureConverter.
toFahrenheit(37.0);
        System.out.println("Celsius: " + celsius);
        System.out.println("Fahrenheit: " +
fahrenheit);
    }
}
```

Key Takeaways

- Class methods are called using the class name and are shared among all instances of the class.
- Instance methods require an object of the class to be called.
- Static methods are useful for operations that do not depend on instance data.

Key Terms

class method

A method associated with the class, called using the class name.

instance method

A method associated with an object, called on an instance of the class.

static

The keyword used to define class methods.

dot operator

The . operator used to call methods on objects or classes.

LESSON 11

Math Class

AP Learning Objectives

1.11.A Develop code to write expressions that incorporate calls to built-in mathematical libraries and determine the value that is produced as a result.

The `Math` class in Java provides a set of **static methods** to perform common mathematical operations, such as calculating square roots, powers, absolute values, calculating square roots, powers, trigonometric operations, and finding the maximum or minimum of two numbers.

This class helps you solve problems efficiently without the need for manually implementing complex formulas. These methods are called using the class name and the dot operator, without the need to create an object. By utilizing the built-in methods of the `Math` class, programmers can simplify complex mathematical calculations, avoid writing custom algorithms, and make their code more efficient and readable.

This lesson emphasizes how to use methods like `Math.sqrt()`, `Math.pow()`, `Math.abs()`, and others to solve real-world problems easily within Java programs.

Calling Math class methods

Because the methods in the `Math` class are static, they are called using the class name, not on objects. This is achieved using the **dot operator** (e.g., `Math.sqrt(value)`).

Example:

Here are some commonly used methods in the `Math` class:

`Math.sqrt(double value)`: Returns the square root of the value.

```
| double result = Math.sqrt(16); // Result is 4.0
```

`Math.pow(double base, double exponent)`: Raises a base to a given power.

```
| double result = Math.pow(2, 3); // Result is 8.0 (2^3)
```

`Math.abs(int value)`: Returns the absolute value of an integer or floating-point number.

```
| int absoluteValue = Math.abs(-10); // Result is 10
```

`Math.max(int a, int b)` and `Math.min(int a, int b)`: Returns the maximum or minimum of two numbers.

```
| int maxValue = Math.max(10, 20); // Result is 20
| int minValue = Math.min(5, 3);   // Result is 3
```

`Math.random()`: Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

```
| double randomValue = Math.random();
| // Possible result: 0.735
```

To get a whole number, multiply the result and cast it to `int`.

```
| int randomInt = (int)(Math.random() * 10);
| // Random int from 0 to 9
| int diceRoll = (int)(Math.random() * 6) + 1;
| // Random int from 1 to 6
```

Activities

1. Explain how the `Math` class in Java helps write cleaner code in Java.
2. Update the Pythagorean theorem Java program from the previous lesson to use the `Math` class methods.
3. Write a Java program that asks the user to input the coordinates of two points in a x,y plane and then calculates the Euclidean distance between those 2 points.
4. Write a Java program that performs the following tasks using methods from the `Math` class:
 1. Calculate and print the square root of 49.
 2. Find the maximum between two numbers, 15 and 30.
 3. Compute the result of 5 raised to the power of 3.
 4. Print the absolute value of -25.

Key Takeaways

- The `Math` class provides static methods for mathematical calculations, such as `sqrt()`, `pow()`, `abs()`, and more.
- You don't need to instantiate the `Math` class to use its methods, as they are static and can be accessed using the dot operator with the class name.
- Using the `Math` class simplifies calculations in programs, improving efficiency and readability.

Key Terms

static methods

Methods that belong to the class rather than an instance of the class.

Math class

A Java class containing methods for performing standard mathematical operations.

LESSON 12

Objects: Instances and Classes

AP Learning Objectives

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.B Develop code to declare variables to store reference types.

Objects and **classes** are fundamental concepts in Java, forming the foundation of object-oriented programming. A class defines the structure and behavior (methods) that its objects will have, acting as a blueprint. An object is a specific **instance** of a class, with its own set of attributes and behaviors. This lesson delves into understanding how objects are created, how they interact with classes, and how they relate to one another in Java. It will help you understand the relationship between classes and objects, learn how to declare and use reference variables to store object references and gain familiarity with how Java organizes classes and the concept of class hierarchies.

Objects and Classes

A **class** is a blueprint that defines the characteristics and behaviors that its objects will have. These characteristics are stored in attributes, and behaviors are represented through methods. An **object** is an actual instance of a class with its own data.

Example:

```
// This is the class definition for a Rectangle
class Rectangle {
    int width;
    int height;
```

```

// Constructor to initialize object
Rectangle(int w, int h) {
    width = w;
    height = h;
}

// Method to calculate the area of the rectangle
int calculateArea() {
    return width * height;
}
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of class Rectangle
        Rectangle rect1 = new Rectangle(5, 10);

        int area1 = rect1.calculateArea();
        // Calls the calculateArea method
        System.out.println("Area of rectangle 1: " +
area1);
    }
}

```

Output:

```
Area of rectangle 1: 50
```

In this example, the class `Rectangle` defines two attributes (`width` and `height`) and a method `calculateArea()` that calculates the area of the rectangle. The object `rect1` is created from the `Rectangle` class and initialized with specific values (a width of 5 and a height of 10).

The **new operator** in Java is used to create new objects or instances of a class. When you use the **new** operator, memory is allocated for the object, and the class constructor is called to initialize the object's fields (attributes).

How the **new** Operator works:

1. **Allocates Memory:** The **new** operator allocates space in memory for a new object.
2. **Calls the Constructor:** Once memory is allocated, the **new** operator calls the constructor of the class. The constructor is responsible for initializing the object, assigning initial values to its fields.

- 3. Returns a Reference:** After creating and initializing the object, the `new` operator returns a reference to the memory location where the object is stored. This reference is then assigned to a **reference variable**, allowing you to interact with the object.

Syntax:

```
ClassName objectName = new ClassName(arguments);
```

- `ClassName`: The name of the class for which you want to create an object.
- `objectName`: The reference variable that will hold the memory address of the newly created object.
- `new ClassName(arguments)`: The `new` operator followed by the constructor call. The arguments passed to the constructor initialize the object.

More examples using the new Operator:

```
class Car {
    int year;
    int maxSpeed;

    // Constructor to initialize the fields
    Car(int y, int s) {
        year = y;
        maxSpeed = s;
    }

    // Method to display the car's information
    void displayInfo() {
        System.out.println(year);
        System.out.println(maxSpeed);
    }
}

public class Main {
    public static void main(String[] args) {
        /* Using the 'new' operator to create an object of
        the Car class */
    }
}
```

```

        Car myCar = new Car(2020, 180);

        // Calling a method on the created object
        myCar.displayInfo();
    }
}

```

Output:

```

2020
180

```

Explanation:

- `new Car(2020, 180)`: This creates a new `Car` object with the `year` set to 2020 and the `maxSpeed` set to 180. The `new` operator allocates memory and calls the `Car` constructor.
- `Car myCar`: The reference variable `myCar` holds the memory address of the newly created `Car` object.
- `myCar.displayInfo()`: Using the dot operator, we can call the `displayInfo()` method on the `myCar` object to display its information.

Multiple Objects:

You can use the `new` operator multiple times to create as many objects as needed from the same class, each with its own independent state (attributes).

Example of Multiple Object Creation:

```

public class Main {
    public static void main(String[] args) {
        /* Creating multiple objects using the 'new'
        operator */
        Car car1 = new Car(2015, 150);
        Car car2 = new Car(2021, 200);

        // Displaying the details of both cars
        car1.displayInfo();
        car2.displayInfo();
    }
}

```

Output:

```
2015
150
2021
200
```

In this example, two `Car` objects are created with different values for year and speed. Each object is independent of the other, but both are created using the `new` operator.

Reference Variables

A **reference variable** stores the address of an object, not the object itself. When you declare a class object, the reference variable holds the memory location — or address — where the object is stored. This address is like a pointer to where the actual data lives in memory.

Example:

```
Rectangle rect2 = new Rectangle(7, 3);
// rect2 holds a reference to a Rectangle object
```

Here, `rect2` is a reference variable that points to the memory location of a `Rectangle` object initialized with width 7 and height 3.

Object Hierarchies and Inheritance

Though inheritance is not covered in detail here, it is important to note that all classes in Java indirectly inherit from the base class `Object`. This inheritance allows classes to share behavior and attributes from a parent class. For example, methods like `toString()` and `equals()` are inherited from `Object` and are available in every class, even if you don't explicitly define them.

Activities

1. Explain the conceptual difference between classes and objects in Java.
2. Describe how the `new` operator works when called to create an instance of an object.
3. Explain how reference variables for objects differ from general variables.
4. Write a complete Java program that models circles using a class.

Your program should include the following:

1. A class named `Circle`, which has:
 - One attribute: `radius` (of type `double`)
 - A constructor to initialize the radius
 - A method named `getArea()` that calculates and returns the area of the circle using the formula: $\text{area} = 3.14 * \text{radius} * \text{radius}$
2. In the main method:
 - Create two `Circle` objects with different radius values
 - Call the `getArea()` method on each object
 - Print the area of both circles
5. Write a Java program that creates an object of the `Geometry` class.

This class will implement the Pythagorean theorem and Euclidean distance methods you have created in the exercises 2 and 3 of lesson 11.

6. Identify the errors in the following Java code and correct the code to ensure it runs properly:

```
public class Rectangle {  
    int width;  
    int height;  
  
    public Rectangle(int w, int h) {  
        width = w;  
        height = h;  
    }  
  
    public int calculateArea() {  
        return width * height;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Rectangle rect;  
        rect.width = 5;  
        rect.height = 10;  
  
        int area = calculateArea();  
        System.out.println("Area: " + area);  
    }  
}
```

Key Takeaways

- A class defines the structure and behavior of objects, while objects are actual instances with specific attribute values.
- Reference variables store the address of the object, allowing access to its data and methods.
- Constructors initialize objects with specific attribute values when they are created.
- Objects of the same class can have different attribute values, even though they share the same structure and behavior.

Key Terms

class

A blueprint for creating objects, defining attributes (fields) and behaviors (methods).

object

An instance of a class that has its own values for the attributes defined in the class.

reference variable

A variable that holds the memory location (address) of an object.

constructor

A method that is automatically called when an object is created, used to initialize the object's attributes.

instance

A concrete occurrence of a class, representing an object in memory.

LESSON 13

Object Creation and Storage (Instantiation)

AP Learning Objectives

1.13.A Identify, using its signature, the correct constructor being called.

1.13.B Develop code to declare variables of the correct types to hold object references.

1.13.C Develop code to create an object by calling a constructor.

In Java, object creation and instantiation involve using constructors to initialize objects. A class contains constructors that have the same name as the class, and these constructors allow objects to be created. The `new` keyword is used to call a constructor and allocate memory for the object. Understanding the role of constructors, how to pass arguments to them, and how the flow of control shifts during object creation are essential for managing object-based programming.

Constructor Signatures

A constructor is a special method with the same name as the class. It is responsible for initializing objects when they are created. A constructor does not specify a return type, but it implicitly returns a new instance of the class. A constructor signature consists of the constructor's name and its list of parameter types. The parameter list specifies the types and names of the values passed to the constructor.

Example:

```
public class Car {  
    int speed;
```

```

        // Constructor with a parameter
        public Car(int initialSpeed) {
            speed = initialSpeed;
        }
    }

```

In the above class, the constructor `Car(int initialSpeed)` takes an integer parameter and assigns it to the speed field of the Car object.

Calling Constructors:

Objects are created using the `new` keyword followed by a constructor call. The constructor initializes the object with values, either default or passed as arguments.

Example:

```

Car myCar = new Car(50); /* Creates a Car object with
                        speed set to 50 */

```

This code creates a new `Car` object and stores a reference to it in the variable `myCar`.

Object References:

A reference variable in Java holds the memory address of the object. If no object is assigned, the variable can hold `null`.

Example:

```

Car anotherCar = null; /* Reference variable holding
                        no object */

anotherCar = new Car(70); /* Assigns an object to the
                        reference */

```

Constructor Overloading:

A class can have multiple constructors with different parameter lists. This is known as constructor overloading.

Example:

```

public class Car {
    private int speed;
    private String color;
}

```

```
// Overloaded constructors
public Car(int initialSpeed) {
    speed = initialSpeed;
}

public Car(int initialSpeed, String carColor) {
    speed = initialSpeed;
    color = carColor;
}
}
```

In this case, you can create a Car object using either one or two parameters:

```
Car fastCar = new Car(100);
Car redCar = new Car(80, "Red");
```

Call by Value:

When calling a constructor, arguments are passed by value. This means that the constructor works with copies of the passed arguments, not the original variables.

Example:

```
int speed = 60;
Car myCar = new Car(speed); // speed is passed by value
```

Flow of Control:

When a constructor is called, the program pauses normal execution to execute the constructor's statements. After the constructor finishes, control returns to the point immediately after the object was instantiated.

Examples

```
// Defining a simple class with constructors
class Rectangle {
    int width, height;

    // Constructor without parameters (default)
    public Rectangle() {
        width = 1;
    }
}
```

```

        height = 1;
    }

    // Constructor with two parameters
    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
}

// Using constructors
public class TestRectangle {
    public static void main(String[] args) {
        Rectangle defaultRectangle = new Rectangle(); //
        Calls default constructor
        Rectangle largeRectangle = new Rectangle(5, 10);
        // Calls overloaded constructor
    }
}

```

In this example, Rectangle objects are created using different constructors depending on whether arguments are passed or not.

Activities

1. Write a Java program to create a `Book` class with attributes for the number of pages and the year it was published.

Write two constructors: one that sets default values and another that accepts parameters to initialize these attributes.

2. Identify the errors in the following Java code and correct the code to ensure it runs properly:

```
public class Book {  
    private String title;  
    private String author;  
  
    public Book(String t, String a) {  
        title = t;  
        author = a;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Book myBook;  
        myBook.title = "Java Programming";  
        myBook.author = "John Doe";  
  
        System.out.println("Title: " + title);  
        System.out.println("Author: " + author);  
    }  
}
```

3. Identify the errors in the following Java code and correct the code to ensure it runs properly:

```
public class Laptop {
    private String brand;
    private int memory;

    public Laptop(String b, int m) {
        brand = b;
        memory = m;
    }

    public void displayInfo() {
        System.out.println("Brand: " + brand);
        System.out.println("Memory: " + memory +
"GB");
    }
}

public class Main {
    public static void main(String[] args) {
        Laptop myLaptop = null;
        myLaptop.displayInfo();
    }
}
```

4. Identify the errors in the following Java code and correct the code to ensure it runs properly:

```
public class MathUtils {
    public static int square(int num) {
        return num * num;
    }
}
```

```

public class Main {
    public static void main(String[] args) {
        MathUtils math = new MathUtils();
        int result = math.square(4);
        System.out.println("Square: " + result);
    }
}

```

5. Complete the following Java code so that it successfully creates Student objects and prints their information.

```

public class Student {
    private String name;
    private int gradeLevel;

    // Constructor to initialize student information
    public Student(_____, _____) {
        // (A) COMPLETE THE PARAMETERS

        _____ = _____;
        // (B) ASSIGN PARAMETERS TO INSTANCE VARIABLES
        _____ = _____;
    }

    // Method to display student details
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Grade Level: " +
gradeLevel);
    }
}

public class Main {
    public static void main(String[] args) {

```

```

        // Creating Student objects
        Student student1 = new Student(_____,
        _____);
        // (C) PASS ARGUMENTS
        Student student2 = new Student(_____,
        _____);

        // Display student information
        student1._____( );
        // (D) CALL METHOD
        student2._____( );
    }
}

```

6. Complete the following Java code so that it supports constructor overloading.

```

public class Car {
    private int speed;
    private String color;

    // Constructor that initializes only speed
    public Car(_____) {
        // (A) COMPLETE THE PARAMETER LIST
        this.speed = _____;
        // (B) ASSIGN VALUE TO speed
    }

    // Overloaded constructor that initializes speed
    // and color
    public Car(_____, _____) {
        // (C) COMPLETE THE PARAMETER LIST
        _____ = _____;
        // (D) ASSIGN VALUES TO INSTANCE VARIABLES
        _____ = _____;
    }
}

```

```

        // Method to display car details
        public void displayInfo() {
            System.out.println("Speed: " + speed);
            System.out.println("Color: " + color);
        }
    }

    public class Main {
        public static void main(String[] args) {
            // Creating objects using different
            constructors

            Car car1 = new Car(______);
            // (E) PASS ONLY SPEED

            Car car2 = new Car(______, ______);
            // (F) PASS SPEED AND COLOR

            // Display car information
            car1.displayInfo();
            car2.displayInfo();
        }
    }
}

```

Key Takeaways

- Objects in Java are created using constructors, which initialize the object's attributes.
- A constructor can be overloaded to provide multiple ways to initialize an object.
- Reference variables hold the memory address of objects, and the `new` keyword allocates memory for new objects.
- Passing arguments to constructors is done by value, meaning that copies of the arguments are used within the constructor.

Key Terms

constructor

A special method used to initialize objects.

overloading

Having multiple methods or constructors with the same name but different parameter lists.

object reference

A variable that holds the memory address of an object.

call by value

The method of passing copies of arguments to constructors or methods.

LESSON 14

Calling Instance Methods

AP Learning Objectives

1.14.A Develop code to call instance methods and determine the result of these calls.

In this lesson, we focus on calling instance methods in Java. Instance methods define behaviors that are specific to objects created from a class. These methods can interact with the attributes of the object and perform actions based on the object's state. The dot operator is used to call these methods on specific objects. An understanding of how instance methods are called is essential for manipulating and interacting with objects effectively.

Calling Instance Methods

Instance methods are specific to objects created from a class and are called using the **dot operator** (.). The general format for calling an instance method is:

```
objectName.methodName();
```

Here, `objectName` is the name of the object, and `methodName` is the method to be called.

Example:

```
class Car {  
    public void startEngine() {  
        System.out.println("Engine started!");  
    }  
}  
  
public class Main {
```

```

    public static void main(String[] args) {
        Car myCar = new Car(); // Create a new Car object
        myCar.startEngine();
        // Call the startEngine method on the Car object
    }
}

```

Output:

```
Engine started!
```

In this example, `myCar` is an instance of the `Car` class, and the `startEngine` method is called using the dot operator.

NullPointerException

If an attempt is made to call a method on a null reference (i.e., an object that has not been properly initialized), a `NullPointerException` is thrown. This is a common error when working with objects in Java.

Example:

```

Car myCar = null;
myCar.startEngine(); /* This will throw a
                     NullPointerException */

```

Activities

1. Describe what instance methods are in Java
2. Explain what causes `NullPointerException` errors and how understanding them can help you write safer, cleaner code.
3. Describe the main differences between instance methods and static methods in Java.
4. Write a program that creates an instance of a `Calculator` class.

The class should have a method named `addNumbers` that adds two integers and prints the result. Call the method using the dot operator on the created object.

5. Identify the errors in the following Java code and correct the code to ensure it runs properly:

```
public class BankAccount {  
    private double balance;  
  
    // Constructor to initialize balance  
    public BankAccount(double initialBalance) {  
        balance = initialBalance;  
    }  
  
    // Instance method to deposit money  
    public void deposit(double amount) {  
        balance += amount;  
        System.out.println("Deposited: $" + amount);  
    }  
  
    // Static method to withdraw money
```

```
public static void withdraw(double amount) {
    balance -= amount;
    System.out.println("Withdrawn: $" + amount);
}

// Instance method to display balance
public void displayBalance() {
    System.out.println("Current Balance: $" +
balance);
}

public static void main(String[] args) {
    BankAccount myAccount = new BankAccount(1000);

    myAccount.deposit(200);

    myAccount.withdraw(100);

    myAccount.displayBalance();
}
}
```

Key Takeaways

- Instance methods are invoked using the dot operator, and they operate on specific objects.
- Ensure that an object is properly initialized before calling methods on it to avoid `NullPointerException`.
- Instance methods allow objects to perform specific actions based on their state.

Key Terms

instance method

A method that is called on an instance of a class.

NullPointerException

An exception that occurs when trying to call a method or access a field on a null object reference.

LESSON 15

String Manipulation

AP Learning Objectives

1.15.A Develop code to create string objects and determine the result of creating and combining strings.

1.15.B Develop code to call methods on string objects and determine the result of calling these methods.

In Java, **strings** are sequences of characters used for storing and manipulating text. The `String` class, part of the `java.lang` package, provides a variety of methods for working with string objects. Strings in Java are **immutable**, meaning that once created, their value cannot be changed. However, you can perform various operations on strings, such as **concatenation**, slicing (substrings), and comparisons. Understanding how to manipulate strings is a fundamental skill for solving problems in Java, as strings are frequently used for input, output, and data processing.

String fundamentals

- A `String` object represents a sequence of characters and can be created by using a string literal.
- The `String` class is part of the `java.lang` package. Classes in this package are available by default.
- A `String` object is **immutable**, meaning once it is created, its value cannot be changed. Methods that act on a `String` object return new `String` objects without altering the original.
- In Java, you can create a new `String` object using a constructor with the `new` keyword. This explicitly creates a new `String` object in memory, even if the content is the same as an existing string.

- Two `String` objects can be **concatenated** using the `+` or `+=` operator, which results in a new `String` object. When a primitive value is concatenated with a `String`, the primitive value is implicitly converted to a string.
- A `String` can be concatenated with any object, which implicitly calls the object's `toString` method. The `toString` method provides a string representation of the object.
- A `String` object has index values starting at 0 and going up to `length() - 1`. Attempting to access an invalid index will result in an `IndexOutOfBoundsException`.
- Common `String` methods include:
 - > `int length()`: Returns the number of characters in the string.
 - > `String substring(int from, int to)`: Returns a substring starting from index `from` and ending at `to - 1`.
 - > `int indexOf(String str)`: Returns the index of the first occurrence of the specified substring. Returns `-1` if the substring is not found.
 - > `boolean equals(Object other)`: Compares two strings for equality.
 - > `int compareTo(String other)`: Compares two strings lexicographically.

String Manipulation in Detail

Creating Strings

In Java, strings can be created by assigning a string literal to a `String` variable:

```
String greeting = "Hello, World!";
```

This creates a `String` object representing the text `"Hello, World!"`. Since strings are immutable, this object cannot be modified once created. Any operations on `greeting` will produce new strings.

String Concatenation

Concatenation allows you to combine two strings into one. This can be done using the `+` operator or the `+=` operator.

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;
// Result: "John Doe"
```

Here, `firstName` and `lastName` are concatenated, and a space is inserted between them. The result is a new `String` object with the value `"John Doe"`.

Concatenation can also involve primitive types:

```
int age = 25;
String message = "Age: " + age; // Result: "Age: 25"
```

When a primitive type (like `int`) is concatenated with a `String`, Java automatically converts the primitive to a string.

String Methods

Java's `String` class provides several useful methods for manipulating strings:

- `length()`: Returns the number of characters in the string.

```
String text = "Hello";
int length = text.length(); // Result: 5
```

- `substring(int from, int to)`: Returns a substring starting from index `from` and ending at `to - 1`.

```
String text = "Hello, World!";
String sub = text.substring(7, 12); // Result: "World"
```

- `indexOf(String str)`: Returns the index of the first occurrence of the specified substring. If the substring is not found, it returns `-1`.

```
String text = "Hello, World!";
int index = text.indexOf("World"); // Result: 7
```

- `equals(Object other)`: Compares two strings for equality.

```
String a = "Hello";
String b = "Hello";
boolean areEqual = a.equals(b); // Result: true
```

- `compareTo(String other)`: Compares two strings lexicographically meaning in dictionary order.

```
String a = "Apple";
String b = "Banana";
int result = a.compareTo(b);
// Result: negative because "Apple" is less than "Banana"
```

- `toUpperCase()`: Converts all characters in the string to uppercase.

```
String text = "hello";
String upper = text.toUpperCase(); // Result: "HELLO"
```

- `toLowerCase()`: Converts all characters in the string to lowercase.

```
String text = "HELLO";
String lower = text.toLowerCase(); // Result: "hello"
```

- `charAt(int index)`: Returns the character at the specified index (starting from 0).

```
String text = "World";
char letter = text.charAt(1); // Result: 'o'
```

Common Exceptions

- `IndexOutOfBoundsException`: This occurs when you try to access an index that is outside the range of the string.

```
String text = "Hello";
char invalidChar = text.charAt(10);
// This will throw IndexOutOfBoundsException
```

Example: String Manipulation

```
public class StringManipulation {
    public static void main(String[] args) {
        String text = "Programming in Java";

        // Length of the string
        System.out.println("Length: " + text.length());
    }
}
```

```
// Extracting a substring
String language = text.substring(15);
System.out.println("Language: " + language);

// Finding the index of a word
int index = text.indexOf("Java");
System.out.println("Index of 'Java': " + index);

// Concatenation
String message = "Learning " + language;
System.out.println(message);
    }
}
```

Output:

```
Length: 19
Language: Java
Index of 'Java': 15
Learning Java
```

Activities

1. Explain the term immutable to describe strings in Java.
2. Describe the benefits of Java using the `String` class to operate on strings instead of it being a built-in data type like `int` and `double`.
3. Write a Java program that asks the user for their full name and displays their initials.

For example, if the user enters "John Doe", the program should display "JD".

4. What is the output of the following Java program:

```
public class StringTest {  
    public static void main(String[] args) {  
        String phrase = "Testing text manipulation  
methods";  
  
        // Extracting a substring  
        String subject = phrase.substring(3, 11);  
  
        // Finding the index of a word  
        int index = phrase.indexOf("text");  
  
        // Concatenation  
        String message = subject + " " + "is the  
subject";  
  
        System.out.println("Subject: " + subject);  
        System.out.println("Index of text: " + index);  
        System.out.println(message);  
    }  
}
```

5. What is the output of the following Java program:

```
public class StringComparison {
    public static void main(String[] args) {
        String word1 = "Hello";
        String word2 = new String("Hello");
        String word3 = "Java";

        // String length
        int length1 = word1.length();
        int length2 = word3.length();

        // Equality check
        boolean isEqual = word1.equals(word2);

        // Lexicographical comparison
        int compareResult = word1.compareTo(word3);

        System.out.println("Length of word1: " +
            length1);
        System.out.println("Length of word3: " +
            length2);
        System.out.println("Are word1 and word2 equal?
            " + isEqual);
        System.out.println("Lexicographical comparison
            (word1 vs word3): " + compareResult);
    }
}
```

6. A document processing system needs an initials extractor to display a person's initials from their full name.

The program should:

- Convert the name to uppercase.
- Extract the first letter of the first name.
- Extract the first letter of the last name.
- Concatenate them with "." in between.

Your task is to write a program that extracts and displays the initials.

7. An email service needs a formatter that standardizes email addresses. The system should:

- Convert the username to lowercase.
- Remove extra spaces around the username.
- Replace spaces within the username with underscores (_).
- Ensure the domain is always "@school.edu".

Your task is to write a Java program that takes a user's name and generates a standardized email.

Key Takeaways

- Strings in Java cannot be changed once created. Operations on strings return new objects.
- Common string methods include `length()`, `substring()`, `indexOf()`, `equals()`, and `compareTo()`.
- Accessing invalid indices in a string will result in an `IndexOutOfBoundsException`.
- Knowing how to manipulate strings is crucial for data handling and processing in Java programs.

Key Terms

string

A sequence of characters.

immutable

A property of strings meaning they cannot be changed after creation.

concatenation

Combining two or more strings into one.

index

The position of a character in a string, starting at 0.

`substring()`

A method that extracts part of a string.

`indexOf()`

A method that returns the position of a substring.

Exam Questions

1. What is the correct way to declare a String variable in Java?

- (A) string name = "Java";
- (B) String name = "Java";
- (C) String name = new String("Java");
- (D) Both (B) and (C)

2. What will be printed by the following code snippet?

```
String str1 = "Hello";  
String str2 = new String("Hello");  
System.out.println(str1 == str2);
```

- (A) true
- (B) false
- (C) HelloHello
- (D) null

3. What will be the output of the following code?

```
String str = "abcdef";  
System.out.println(str.substring(2, 5));
```

- (A) abc
- (B) cde
- (C) Compilation error
- (D) bcde

4. Which of the following statements is true regarding Java's String class?

- (A) Strings must be declared using the new keyword.
- (B) The String class is part of java.util package.
- (C) Strings are immutable.
- (D) The == operator compares the contents of two strings.

5. What will be the output of the following code?

```
String str = "Java";  
str = str.concat(" Rocks!");  
System.out.println(str);
```

- (A) Java Rocks!
- (B) Java
- (C) Compilation Error
- (D) JavaRocks!

6. Which of the following is NOT a method of the String class?

- (A) length()
- (B) charAt(int index)
- (C) compareTo(String anotherString)
- (D) reverse()

7. What is the output of the following code?

```
String str = "Java";  
str.toUpperCase();  
System.out.println(str);
```

- (A) JAVA
- (B) Java
- (C) null
- (D) Exception Thrown

8. What will be the result of the following expression?

```
String s1 = "AP";  
String s2 = "CS";  
String s3 = s1 + s2;  
System.out.println(s3 == "APCS");
```

- (A) true
- (B) false
- (C) Compilation Error
- (D) null

9. What does the following code output?

```
String str = "AP Computer Science A";  
System.out.println(str.indexOf("Computer"));
```

- (A) 0
- (B) 3
- (C) -1
- (D) Compilation Error

10. What will be printed by the following code?

```
String x = "code";  
String y = "Code";  
System.out.println(x.equal(y));
```

- (A) true
- (B) false
- (C) Compilation Error
- (D) Exception Thrown

11. What is the correct way to declare and initialize a primitive integer variable in Java?

- (A) Integer x = new Integer(10);
- (B) int x = 10;
- (C) double x = 10;
- (D) var x = 10;

12. Which of the following statements about Java primitive types is true?

- (A) Primitive types are objects.
- (B) Primitive types have built-in methods.
- (C) Primitive types must be instantiated using new.
- (D) Primitive types are stored directly in memory.

13. What will be the output of the following code?

```
int a = 10;  
int b = 3;  
System.out.println(a / b);
```

- (A) 3.3333
- (B) 3.0
- (C) 3
- (D) Compilation error

14. Which of the following statements about Java objects is correct?

- (A) All Java objects must be declared using static.
- (B) Java objects are automatically garbage collected when they are no longer referenced.
- (C) Java objects can only be created using a constructor without parameters.
- (D) Java objects cannot have instance variables.

15. What is the purpose of a constructor in Java?

- (A) To destroy an object
- (B) To define a new class
- (C) To initialize a new object
- (D) To create new primitive data types

16. What is the output of the following code?

```
int x = 5;  
double y = 2;  
System.out.println(x * y);
```

- (A) 10
- (B) 10.0
- (C) 2.5
- (D) Compilation error

17. Given the following class definition, what will the following code print?

```
public class Dog {  
    private int age;  
  
    public Dog(int a) {  
        age = a;  
    }  
  
    public void increaseAge() {  
        age++;  
    }  
  
    public int getAge() {  
        return age;  
    }  
}
```

```
}  
  
Dog d = new Dog(3);  
d.increaseAge();  
System.out.println(d.getAge());
```

- (A) 3
- (B) 4
- (C) Compilation error
- (D) Runtime Exception

18. Which of the following statements about method overloading in Java is true?

- (A) Overloaded methods must have the same parameter list.
- (B) Overloaded methods must have different method names.
- (C) Overloaded methods must have the same return type.
- (D) Overloaded methods must have different parameter lists.

19. What will be the result of the following expression?

```
double result = Math.sqrt(25);  
System.out.println(result);
```

- (A) 25
- (B) 5.0
- (C) 5
- (D) Compilation error

20. Consider the following code:

```
public class Counter {  
    private int count;  
  
    public Counter() {  
        count = 0;  
    }  
  
    public void increment() {  
        count++;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}  
  
Counter c = new Counter();  
System.out.println(c.getCount());
```

What will be printed?

- (A) 0
- (B) 1
- (C) null
- (D) Compilation error