STUDY NOTES

Unit 1 — Using Objects and Methods

15–25% AP Exam Weighting

CONTENTS

1.1 INTRODUCTION TO ALGORITHMS, PROGRAMMING, AND COMPILERS

An **algorithm** is a finite, ordered sequence of instructions that accomplishes a task. A **program** is an algorithm written in a language a computer can eventually execute. Java programs are written as human-readable source code, which cannot run directly — it must be translated.

- A **compiler** translates source code into a lower-level form, checking for syntax errors along the way. In Java, the compiler turns .java files into **bytecode** (.class files).
- The **Java Virtual Machine (JVM)** then executes that bytecode. This is why Java is "write once, run anywhere" — the same bytecode runs on any machine with a JVM.
- **Syntax errors** are caught at compile time (broken grammar). **Logic errors** slip past the compiler because the code runs but produces the wrong result. **Runtime errors** occur during execution (e.g. dividing by zero).

> **Why it matters:** "Does it compile?" and "Does it run correctly?" are two different questions — the compiler only guarantees the first one.

PRACTICE

1. A program compiles successfully but always prints the wrong average. What type of error is this, and why didn't the compiler catch it?

*Answer: This is a **logic error**. The compiler only checks Java's grammar rules (syntax) — it can't know what result you intended, so incorrect logic still compiles and runs.*

2. Put these in order: bytecode execution, writing source code, compiling.

Answer: 1) Writing source code → 2) Compiling (source → bytecode) → 3) Bytecode execution (by the JVM).

1.2 Variables and Data Types

A **variable** is a named piece of memory that holds a value. Every variable in Java has a fixed **data type**, declared once, that determines what kind of value it can store and how much memory it uses.

Type	Stores	Example
int	Whole numbers	int age = 17;
double	Decimal numbers	double gpa = 3.85;
boolean	true / false	boolean isDone = false;
String	Text (not primitive — a class)	String name = "Ada";

A variable must be **declared** (type + name) before it's used, and typically **initialized** (given a starting value) in the same step.

```
int score = 0;
double price = 19.99;
boolean passed = true;
String course = "AP CSA";
```

PRACTICE

1. Which data type would you use to store the price of an item like $4.50, and which for the number of items in a cart?

   Answer: *double for the price ($4.50 has a fractional part); int for the item count (whole numbers only).*

2. Is this valid Java? What is this pattern called?

```
JAVA
int total;
total = 25;
```

   Answer: *Yes, it's valid. Line 1 is the **declaration** (reserves memory, sets the type); line 2 is a separate **assignment** that gives it a value.*

1.3 Expressions and Output

An **expression** is any combination of values, variables, and operators that evaluates to a single value. Java uses the standard arithmetic operators: + - * / %.

- / between two ints performs **integer division** — the decimal part is truncated (7 / 2 is 3, not 3.5).
- % (modulo) returns the **remainder** of division (7 % 2 is 1).
- Operator precedence follows normal math rules: * / % before + -, evaluated left to right; use parentheses to force order.

```
JAVA
System.out.println(7 / 2);      // 3  (integer division)
System.out.println(7 % 2);      // 1  (remainder)
System.out.println(7.0 / 2);    // 3.5 (a double is involved)
System.out.print("Score: ");
System.out.println(95);         // print vs println: println adds a newline
```

System.out.print() and System.out.println() send output to the console; println moves to a new line afterward, print does not.

PRACTICE

1. What does System.out.println(10 % 3 + 1); print?

   Answer: *2 — 10 % 3 is 1 (remainder), then 1 + 1 = 2.*

2. Why does 5 / 2 evaluate to 2 instead of 2.5?

   Answer: *Both operands are int literals, so Java performs integer division and truncates the decimal — it doesn't round.*

1.4 Assignment Statements and Input

The **assignment operator** = stores the value of the right-hand expression into the variable on the left. The right side is fully evaluated first.

```java
int x = 5;
x = x + 1;      // reads old x (5), computes 6, stores it back into x
System.out.println(x); // 6
```

To get input from the user, Java commonly uses the Scanner class from java.util:

```java
import java.util.Scanner;

Scanner in = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = in.nextInt();     // nextDouble(), nextLine() also exist
```

Common mix-up: = assigns a value; == compares two values. This is a frequent source of bugs later on.

PRACTICE

1. After this runs, what is the value of b?

```java
int a = 4;
int b = a;
a = 10;
```

Answer: *4. b = a; copies the value 4 at that moment; changing a afterward doesn't affect b.*

2. Which Scanner method would you use to read a decimal number typed by the user?
Answer: *nextDouble()*

1.5 Casting and Range of Variables

Every numeric type has a limited **range** of values it can hold (e.g. int typically spans about ±2.1 billion). **Casting** explicitly converts a value from one type to another.

```java
double pi = 3.9;
int whole = (int) pi;      // 3 — truncates, does NOT round

int a = 7, b = 2;
double result = (double) a / b;  // 3.5 — cast forces real division
```

- **Narrowing** (double → int) loses information and needs an explicit cast.
- **Widening** (int → double) is automatic — no data is lost.

- Casting (int) on a decimal **truncates toward zero**; it never rounds.
- **Integer overflow:** exceeding a type's range wraps around silently (no error/crash) — a classic source of hard-to-find bugs.

PRACTICE

1. What does (int) 9.99 evaluate to? What about (int) -9.99?

   Answer: *9 and -9. Casting to int truncates the decimal part rather than rounding, and truncation moves toward zero.*

2. You want the true decimal average of two int variables sum and count. Write the correct expression.

   Answer: *(double) sum / count — casting before the division forces decimal division. Casting the whole result, e.g. (double)(sum / count), is a common wrong answer because integer division already happened by then.*

1.6 Compound Assignment Operators

Compound operators combine an arithmetic operation with assignment in one step: var = var OP value becomes var OP= value.

Compound	Equivalent to
x += 5;	x = x + 5;
x -= 3;	x = x - 3;
x *= 2;	x = x * 2;
x /= 4;	x = x / 4;
x++;	x = x + 1;
x--;	x = x - 1;

PRACTICE

1. What is score at the end?

```java
int score = 8;
score *= 3;
score -= 4;
```

   Answer: *20. 8 * 3 = 24, then 24 - 4 = 20.*

2. Rewrite total = total % 5; using a compound operator.

   Answer: *total %= 5;*

1.7 Application Program Interface (API) and Libraries

An **API** is a collection of classes and methods someone else already wrote, that you can use without knowing how they're implemented internally. Java's built-in libraries are organized into **packages** (e.g. java.util, java.lang).

```java
import java.util.Scanner;   // brings a class in from a package
import java.util.ArrayList;
```

- Classes in java.lang (like Math, String, System) are available automatically — no import needed.
- Everything else needs an explicit import statement.
- Using an API well means reading its **documentation** to know what a method needs (parameters) and what it gives back (return type) — not reading its source code.

PRACTICE

1. Why don't you need import java.lang.Math; before using Math.sqrt()?

   Answer: *java.lang is imported automatically into every Java program, since it contains fundamentals like Math, String, and System.*

2. What's the practical benefit of an API — why not just write everything yourself?

   Answer: *It lets you reuse well-tested, efficient code without re-implementing or even understanding its internals — you only need to know its inputs and outputs.*

1.8 Documentation with Comments

Comments are ignored by the compiler and exist purely to explain code to humans.

```java
// single-line comment

/* multi-line
   comment */

/**
 * Javadoc comment — describes a class or method
 * @param n the number to square
 * @return the square of n
 */
public static int square(int n) {
    return n * n;
}
```

Good comments explain **why**, not just restate **what** the code already says clearly. Javadoc-style comments (/** ... */) placed above a method are used to auto-generate documentation and are the standard way to document parameters and return values.

PRACTICE

1. Between a comment that just restates the code and one that explains why the increment happens — which is more useful, and why?

   Answer: *The one explaining why. Restating what the code already shows adds no value; explaining the reasoning conveys information the code itself can't.*

2. What Java tag would you use inside a Javadoc comment to describe what a method returns?

   Answer: *@return (with @param used for each parameter).*

1.9 Method Signatures

A method's **signature** is its name plus its parameter list — it's what the compiler uses to identify which method you're calling. The full header adds the access level and return type:

JAVA

```
public static double
average(int a, int b) {
    return (a + b) / 2.0;
}
```

Part	Example	Meaning
Access modifier	public	Who can call it
static	static	Belongs to the class, not an object
Return type	double	Type of value sent back (or void for none)
Name + parameters	average(int a, int b)	The signature itself

A method with void as its return type performs an action but doesn't send back a value, so it can't be used inside an expression.

PRACTICE

1. What's wrong with this method header? public static void isEven(int n) { return n % 2 == 0; }

   Answer: *It's declared void but tries to return a value. A boolean result requires the return type to be boolean, not void.*

2. Can two methods in the same class both be named print? Under what condition?

   Answer: *Yes — this is **overloading**. It's allowed as long as their parameter lists differ, since the signature must be unique.*

1.10 Calling (Static) Class Methods

A **static** method belongs to the class itself, not to any individual object, so it's called directly through the class name — no object needs to be created first.

```java
ClassName.methodName(arguments);

// concrete example
double root = Math.sqrt(16);   // 4.0
```

- **Parameters** are the placeholders in the method definition; **arguments** are the actual values passed in when you call it.
- Arguments are matched to parameters **by position and type**, in order.
- If the method returns a value, you can store it, print it, or use it directly in another expression.

PRACTICE

1. Write a statement that calls triple(4) and prints the result, given public static int triple(int n) { return n * 3; }.
   Answer: *System.out.println(triple(4)); — this prints 12.*

2. Why can Math.pow(2, 3) be called without creating a Math object first?
   Answer: *Because pow is a **static** method of the Math class — static methods belong to the class as a whole, so they're called through the class name directly.*

1.11 The Math Class

Math lives in java.lang (no import needed) and provides static methods and constants for common numeric operations.

Member	Does
Math.abs(x)	Absolute value
Math.pow(b, e)	b raised to the power e
Math.sqrt(x)	Square root
Math.max(a, b) / Math.min(a, b)	Larger / smaller value
Math.random()	Random double, 0.0 ≤ x < 1.0
Math.PI	The constant π (a field, not a method)

```java
// random integer from 0 to 9 (inclusive)
int die = (int) (Math.random() * 10);

// random integer from 1 to 6 (inclusive) — classic die roll
int roll = (int) (Math.random() * 6) + 1;
```

Member	Does
	Pattern to memorize: (int)(Math.random() * range) + min generates a random int from min to min + range - 1.

PRACTICE

1. Write an expression for a random integer between 10 and 20, inclusive.
   Answer: *(int) (Math.random() * 11) + 10 — the range 10–20 has 11 possible values, so multiply by 11, then shift up by adding the minimum, 10.*

2. What does Math.max(Math.abs(-7), 3) evaluate to?
   Answer: *7. Math.abs(-7) is 7, and Math.max(7, 3) is 7.*

1.12 Objects: Instances of Classes

A **class** is a blueprint that defines what data (fields) and behavior (methods) its objects will have. An **object** is a specific instance built from that blueprint, with its own copy of the fields.

JAVA

```
public class Dog {
    String name;   // instance field — each Dog has its own
    int age;

    public Dog(String n, int a) {   // constructor
        name = n;
        age = a;
    }

    public void bark() {
        System.out.println(name + " says woof!");
    }
}
```

Think of the class Dog as the blueprint, and each individual dog object (like myDog, yourDog) as a house built from it — same structure, different data inside.

PRACTICE

1. If you make two Dog objects with different names, do they share the same name field or have separate copies?
   Answer: *Separate copies. Each object gets its own independent set of instance fields, so changing one object's name doesn't affect the other's.*

2. In the analogy of a class as a blueprint, what does an object represent?
   Answer: *A specific thing built from that blueprint — e.g. one actual house, with its own address and paint color, even though it follows the same plan as every other house built from it.*

1.13 Object Creation and Storage (Instantiation)

Instantiation is the act of creating an actual object from a class, using the new keyword, which calls a **constructor**.

```
JAVA
Dog myDog = new Dog("Rex", 3);
// ^type    ^name   ^new keyword  ^constructor call
```

- Dog myDog declares a variable that can refer to a Dog object.
- new Dog("Rex", 3) actually builds the object in memory and runs its constructor.
- myDog stores a **reference** (essentially an address) to that object — not the object's data directly.

> **Key idea:** object variables hold references. Copying Dog copy = myDog; makes copy point to the same object — it does not create a second dog.

PRACTICE

1. What is a.age after this code runs?

```
JAVA
Dog a = new Dog("Fido", 2);
Dog b = a;
b.age = 9;
```

Answer: *9. b = a; copies the reference, not the object — so a and b point to the same Dog, and changing it through b changes it for a too.*

2. What Java keyword actually triggers memory allocation for a new object?

Answer: *new*

1.14 Calling Instance Methods

Unlike static methods, an **instance method** operates on a specific object's data, so it must be called through an object using dot notation.

```
JAVA
Dog myDog = new Dog("Rex", 3);
myDog.bark();              // "Rex says woof!"

System.out.println(myDog.age);
// accessing a field the same way
```

	Static method	Instance method

Called through	ClassName.method()	object.method()
Operates on	No particular object	That object's own data
Example	Math.sqrt(9)	myDog.bark()

PRACTICE

1. Why does calling bark() require an object like myDog, while calling Math.sqrt() does not?

   Answer: *bark() is an instance method — it depends on that particular dog's name, so Java needs to know which dog. Math.sqrt() is static; it just computes from its argument and needs no object's data.*

2. Given two Dog objects a and b, write a call that makes only b bark.

   Answer: *b.bark(); — the object before the dot determines whose data the method uses.*

1.15 String Manipulation

String is a built-in class (not a primitive) representing text. Strings are **immutable** — once created, a String's characters never change; methods that "modify" a string actually return a brand-new one.

Method	Does	Example
.length()	Number of characters	"hello".length() → 5
.charAt(i)	Character at index i (0-based)	"hello".charAt(1) → 'e'
.substring(a, b)	Characters from a up to (not incl.) b	"hello".substring(1, 4) → "ell"
.indexOf(s)	First index where s begins, or -1	"hello".indexOf("l") → 2
.equals(s)	True if contents match	"hi".equals("hi") → true
+	Concatenation	"a" + "b" → "ab"

JAVA

```
String word = "Computer";
System.out.println(word.length());          // 8
System.out.println(word.substring(0, 3));   // "Com"
System.out.println(word.toUpperCase());     // "COMPUTER"
```

> **Use .equals(), not ==, to compare String contents.** == checks whether two variables refer to the exact same object in memory, which can give surprising, wrong-looking results for equal-looking strings.

PRACTICE

1. What does "programming".substring(3, 7) return? (0-indexed, end exclusive.)
   Answer: *"gram" — characters at index 3, 4, 5, 6 (p-r-o-g-r-a-m-m-i-n-g, indices 0–10).*

2. Why is comparing two Strings with str1 == str2 risky, and what should you use instead?
   Answer: *== compares object references (memory locations), not the text inside. Use str1.equals(str2) to compare actual contents.*

1.16 Operators: Arithmetic, Relational, Logical, and Assignment

Java groups operators into a few families depending on what they act on and what kind of value they produce. This section pulls all four together as one reference.

Arithmetic operators

Work on numbers and produce a number. Covered in detail in 1.3 and 1.5 — summarized here:

Operator	Meaning	Example
+	Addition (also String concatenation)	5 + 2 → 7
-	Subtraction	5 - 2 → 3
*	Multiplication	5 * 2 → 10
/	Division (integer division if both operands are int)	5 / 2 → 2
%	Modulo — remainder after division	5 % 2 → 1

Relational (comparison) operators

Compare two values and always produce a **boolean** (true or false).

Operator	Meaning	Example
==	Equal to	5 == 5 → true
!=	Not equal to	5 != 3 → true
>	Greater than	5 > 3 → true
<	Less than	5 < 3 → false
>=	Greater than or equal to	5 >= 5 → true
<=	Less than or equal to	5 <= 3 → false

Operator	Meaning	Example

> **Careful:** For numbers, == compares values. For objects (including Strings), == compares references, not contents — use .equals() for those, as covered in 1.15.

Logical operators

Combine or invert boolean values.

Operator	Meaning	Example
&&	AND — true only if both sides are true	(5 > 3) && (2 > 1) → true
||	OR — true if at least one side is true	(5 > 3) || (1 > 2) → true
!	NOT — flips a boolean	!(5 > 3) → false

&& and || **short-circuit**: Java stops evaluating as soon as the result is certain. In a && b, if a is false, b is never evaluated at all — this is often used to safely guard against errors.

```java
int[] nums = {1, 2, 3};
int i = 5;

// short-circuiting prevents an out-of-bounds crash:
if (i < nums.length && nums[i] > 0) {
    // safe — nums[i] is never checked because i < nums.length is false
}
```

Assignment operators

Store a value into a variable. The plain = is the base case; the compound forms (see 1.6) fold an arithmetic operation into the assignment.

Operator	Equivalent to	Example (x starts at 10)
=	— (base assignment)	x = 5 → x is 5
+=	x = x + value	x += 3 → x is 13
-=	x = x - value	x -= 3 → x is 7
*=	x = x * value	x *= 2 → x is 20
/=	x = x / value	x /= 2 → x is 5
%=	x = x % value	x %= 4 → x is 2

How they fit together

```java
int age = 20;
```

```
boolean hasLicense = true;

// relational -> boolean, then combined with a logical operator
boolean canDrive = (age >= 16) && hasLicense;   // true

// arithmetic feeding a relational comparison
boolean isEven = (age % 2) == 0;                // true
```

Precedence, roughly high to low: arithmetic (* / % then + -) → relational (< > <= >= == !=) → logical (&& then ||). Parentheses always override this order, and are the easiest way to make intent unambiguous.

PRACTICE

1. What does (10 % 3 == 1) && (10 > 5) evaluate to?

   Answer: *true. 10 % 3 is 1, so the left side 1 == 1 is true; the right side 10 > 5 is also true; true && true is true.*

2. Rewrite price = price - discount; using a compound assignment operator.

   Answer: *price -= discount;*

3. Why is !(a && b) not always the same as !a && !b?

   Answer: *They follow different logic (De Morgan's Law says !(a && b) equals !a || !b, not !a && !b). For example, if a is true and b is false: !(a && b) is !false → true, but !a && !b is false && true → false.*

4. In if (n != 0 && 10 / n > 1), why does the operator order here matter for safety?

   Answer: *&& short-circuits left to right: if n != 0 is false, Java never evaluates 10 / n, avoiding a divide-by-zero error. Swapping the order would remove that protection.*