Lessons 1–12
Console Project

AP Computer Science A — Unit 2 companion (Code.org CSA, The Neighborhood & the Console)

Selection and Iteration

This unit is about how programs **decide** (selection) and **repeat** (iteration). Each section below follows one lesson: the guiding question, the idea, the code, and what to watch for. Keep it open while you work.

Decide

Lessons 2–8: if, if-else, NOT, aliases & equals(), nested ifs, && and ||, De Morgan, else-if

Repeat

Lessons 1, 9, 11: while loops, for loops, nested loops

Apply & analyze

Lessons 10, 12: String algorithms, counting how many times code runs

By the end of this unit you can…

- → Develop and implement algorithms using sequencing, selection, and iteration
- → Use if, if-else, nested conditionals, and while/for loops to solve problems
- → Compare algorithm efficiency using informal runtime analysis
- → Plan and write algorithms using pseudocode and decomposition
- → Write Boolean expressions and apply logical operators to control flow
- → Trace and debug code that uses nested loops and complex conditionals
- → Extend existing classes to create specialized subclasses
- → Collaborate with peers to test and review code

Loops Lesson 1

How can the problem-solving process help me plan and implement an algorithm?

A **while loop** repeats a block of code as long as a condition is true. The condition is a **Boolean expression**: something that is either `true` or `false`. The Painter class gives you Boolean methods that answer questions about the Painter's situation.

Anatomy of a while loop

```
while (myPainter.canMove()) {   // condition checked BEFORE each pass
    myPainter.move();            // body repeats while it's true
}
// loop exits the moment canMove() is false
```

- If the condition is false the first time, the body runs **zero** times.
- Something in the body must eventually make the condition false, or the loop never ends (an **infinite loop**).
- Use a while loop when you don't know in advance how many times to repeat.

Painter Boolean methods

Method	true when…
`canMove()`	the square ahead is open
`isOnBucket()`	standing on a paint bucket
`isOnPaint()`	standing on a painted square
`hasPaint()`	the Painter has paint to use
`isFacingNorth()` etc.	facing that direction

```
while (myPainter.hasPaint()) {
    myPainter.paint("green");
    myPainter.move();
}
```

The problem-solving process: Define the problem → Prepare (break it into smaller tasks, write pseudocode) → Try (code one task at a time) → Reflect (test, fix, repeat). Decomposition means splitting a big job into small methods you can write and test separately.

Selection Statements Lesson 2

How can I write algorithms using sequencing, selection, and iteration in Java?

An **if statement** runs a block of code only when its condition is true. Otherwise the block is skipped and the program continues after it.

```
if (myPainter.isOnBucket()) {
    myPainter.takePaint();      // only when on a bucket
}
myPainter.move();               // always runs
```

Selection inside iteration is the pattern for the whole unit:

```
while (canMove()) {
    if (isOnBucket()) {
        takePaint();
    }
    move();
}
```

The three control structures

Structure	What it does	Java
Sequencing	steps run in order, top to bottom	one statement after another
Selection	choose whether to run a block	`if`, `if-else`, `else if`
Iteration	repeat a block	`while`, `for`

> **Subclass reminder:** when you extend `PainterPlus`, your new class inherits every Painter and PainterPlus method, so you can call `turnRight()` and `canMove()` directly inside your own methods without a dot.

Two-Way Selection Statements Lesson 3

How can I check if a condition is NOT true, or run different code when the condition is false?

if-else: exactly one branch runs

```
if (myPainter.canMove()) {
    myPainter.move();
} else {
    myPainter.turnLeft();       // runs only when canMove() is false
}
```

One of the two blocks always executes, never both, never neither.

NOT (!) flips a Boolean

```
if (!myPainter.hasPaint()) {  // "if NOT hasPaint"
    myPainter.takePaint();
}

while (!myPainter.isOnBucket()) {
    myPainter.move();           // keep going until on a bucket
}
```

a	!a
true	false
false	true

Object Aliases and Equality

Lesson 4

How can I check if two objects are equal?

A primitive variable holds a value. An object variable holds a **reference** (an address).
Two references to the same object are **aliases**.

== on objects asks "same object?"

```
Painter p1 = new Painter();
Painter p2 = new Painter();
Painter p3 = p1;                // p3 is an
alias of p1

p1 == p2      // false: two different objects
p1 == p3      // true: same object, two names
p1 != p2      // true

Painter p4 = null;             // refers to
no object
p4 == null    // true
p4.move();    // NullPointerException!
```

equals() asks "same contents?"

Every class inherits `equals()` from `Object`, but that version just does ==. You **override** it to compare what matters.

```
public class Dessert {
    private String name;
    private double price;

    public boolean equals(Dessert other) {
        return name.equals(other.name)
            && price == other.price;
    }
}
d1.equals(d2)   // true if same name and
price
```

> Primitives: compare with ==. Objects (including Strings): compare with `.equals()`. Use == on objects only to check for aliases or `null`.

Nested If Statements

Lesson 5

How can I test a combination of conditions before executing a specific action?

An if inside an if

```
if (myPainter.isOnBucket()) {
    if (!myPainter.hasPaint()) {   // only tested if on a bucket
        myPainter.takePaint();      // runs only if BOTH are true
    }
}
```

- The inner condition is only checked when the outer one is true.
- An `else` pairs with the **nearest** unmatched `if`. Always use braces so it pairs with the one you mean.
- Tracing tip: indent your trace like the code. Cross out any inner block whose outer condition was false.

String length()

```
String word = "Falcons";
word.length()      // 7
"".length()        // 0

if (word.length() > 5) {
    if (word.substring(0, 1).equals("F")) {
        System.out.println("long F-word");
    }
}
```

The last valid index is always `length() - 1`. Checking length first (outer if) protects the substring call (inner if) from going out of bounds.

Logical Operators

Lesson 6

How can logical operators be used to evaluate multiple conditions?

AND (&&) and OR (||) combine two Boolean expressions into one, which often replaces a nested if.

Truth table

a	b	a && b	a || b
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

- && is true only when **both** sides are true.
- || is true when **at least one** side is true.
- Order of operations: ! first, then &&, then ||. Add parentheses when in doubt.

Same logic, two ways

```
// nested if (Lesson 5)
if (isOnBucket()) {
    if (!hasPaint()) {
        takePaint();
    }
}

// compound expression (Lesson 6)
if (isOnBucket() && !hasPaint()) {
    takePaint();
}
```

Short-circuit evaluation: Java stops as soon as the answer is known. In `a && b`, if `a` is false, `b` is never checked. In `a || b`, if `a` is true, `b` is never checked. That's why `s != null && s.length() > 0` is safe: the length call is skipped when `s` is null.

De Morgan's Laws Lesson 7

What are De Morgan's Laws?

Two Boolean expressions are **equivalent** when they give the same value for every possible input. De Morgan's Laws tell you how to push a NOT through parentheses: distribute the `!` and **flip the operator**.

```
!(a && b)   is equivalent to   !a || !b
!(a || b)   is equivalent to   !a && !b
```

Flip comparisons too when you push a NOT through them:

```
!(x < 5)      →  x >= 5
!(x == y)     →  x != y
!(x > 0 && y > 0)  →  x <= 0 || y <= 0
```

Proving it with a truth table

a	b	!(a && b)	!a || !b
T	T	F	F
T	F	T	T
F	T	T	T
F	F	T	T

Identical columns for all four rows = equivalent expressions. Use this any time a test asks "which expression is equivalent to…"

Multi-Selection Statements Lesson 8

How can I specify different outcomes for different conditions?

```
if (score >= 90) {
    grade = "A";
} else if (score >= 80) {
    grade = "B";
} else if (score >= 70) {
    grade = "C";
} else {
    grade = "F";               // catches everything else
}
```

- Conditions are tested top to bottom; the **first true one wins** and the rest are skipped.
- Because of that, later conditions don't need to repeat earlier ones (`score >= 80` already means "and less than 90").
- The final `else` is optional. Without it, it's possible nothing runs.

else-if vs. separate ifs

```
int x = 95;
if (x > 90) count++;      // runs
if (x > 50) count++;      // ALSO runs → count = 2

if (x > 90) count++;      // runs
else if (x > 50) count++; // skipped → count = 1
```

> **Order matters.** Put the most specific (most restrictive) condition first. If `score >= 70` came first, a 95 would get a "C".

For Loops Lesson 9

How can I write more efficient and readable code using for loops to repeat actions a set number of times?

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

initialize	test	update
runs once	before every pass	after every pass

Output: `012`

3 passes, 4 tests. After the loop, `i` no longer exists.

Execution order

1. int i = 0
2. test i < 3 — false? exit the loop
3. body: print(i)
4. i++, then back to step 2

Reading the range

`i = 0; i < n` → n passes: 0 … n−1

`i = 1; i <= n` → n passes: 1 … n

while → for conversion

```
// while version
int i = 0;
while (i < 4) {
    move();
    i++;
}

// same thing as a for loop
for (int i = 0; i < 4; i++) {
    move();
}
```

Which loop when?

Use a for loop when…	Use a while loop when…
you know how many times (move 5 squares, print 10 rows, visit every index of a String)	you repeat until something happens (until you can't move, until on a bucket, until input is valid)

> Never change the loop variable inside a for loop body. Let the update part do it.

Implementing String Algorithms

Lesson 10

What methods can I use to write algorithms for parsing String objects?

Every String algorithm in this course is a for loop over indexes plus `substring()` to grab one character (or a window of characters) at a time.

Reverse a String

```
String rev = "";
for (int i = 0; i < s.length(); i++) {
    rev = s.substring(i, i + 1) + rev;  // put each char in FRONT
}
```

Palindrome check

```
boolean isPal = s.equals(rev);     // same forwards and backwards
```

Does a String meet a criteria?

```
boolean hasE = false;               // flag starts false
for (int i = 0; i < s.length(); i++) {
    if (s.substring(i, i + 1).equals("e")) {
        hasE = true;                // flip it when found
    }
}
```

Count how many match

```
int vowels = 0;
for (int i = 0; i < s.length(); i++) {
    String ch = s.substring(i, i + 1);
    if (ch.equals("a") || ch.equals("e") || ch.equals("i")
     || ch.equals("o") || ch.equals("u")) {
        vowels++;
    }
}
```

Find a substring (sliding window)

```
String target = "an";
int hits = 0;
for (int i = 0; i <= s.length() - target.length(); i++) {
    if (s.substring(i, i + target.length()).equals(target)) {
        hits++;
    }
}
```

> **The bound is the whole question.** A window of width `w` can start no later than index `length − w`, so the test is `i <= s.length() − w`. One more and you get a StringIndexOutOfBoundsException.

Nested Iteration

Lesson 11

How can I adjust loop control variables of nested loops to produce different output?

Think of a clock: the minute hand (inner loop) goes all the way around once for every single tick of the hour hand (outer loop). The inner loop runs *completely* for each pass of the outer loop.

```
for (int row = 1; row <= 3; row++) {     // outer: hour hand
    for (int col = 1; col <= 4; col++) { // inner: minute hand
        System.out.print("*");
    }
    System.out.println();                // new line after each row
}
```

```
****
****
****
```

Outer controls rows, inner controls what's in each row. Total stars: 3 × 4 = 12.

Inner bound depends on outer variable

```
for (int row = 1; row <= 4; row++) {
    for (int col = 1; col <= row; col++) {
// ← row, not 4
        System.out.print(row);
    }
    System.out.println();
}
```

```
1
22
333
4444
```

row	inner col values	prints
1	1	1
2	1, 2	22
3	1, 2, 3	333
4	1, 2, 3, 4	4444

Tracing tip: make one row in your trace table per outer pass, and list every inner value on that row. Then write what gets printed. Don't try to hold it in your head.

Informal Run-Time Analysis

Lesson 12

How can I determine the number of times a specific statement is executed within a loop?

Fewer statement executions means a faster program. The question on the exam is always "how many times does *this line* run?"

Loop	Body runs
`for (i = 0; i < n; i++)`	n times
`for (i = 1; i <= n; i++)`	n times
`for (i = 3; i < n; i++)`	n − 3 times
`for (i = 0; i < n; i += 2)`	about n / 2 times
nested, bounds n and m	n × m times
two loops in a row, n and m	n + m times
inner bound = outer variable (1..n)	1 + 2 + … + n = n(n+1)/2

Compare two algorithms

```
// A: two separate loops → 2n executions
for (int i = 0; i < n; i++) { count++; }
for (int j = 0; j < n; j++) { count++; }

// B: nested loops → n × n executions
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) { count++; }
}
```

For n = 100: A runs 200 times, B runs 10,000 times. Nested loops get expensive fast.

> **Exam move:** when the count isn't obvious, plug in n = 1, 2, 3, count by hand, and spot the pattern.

Common traps

where points go to die

= vs ==	`=` assigns, `==` compares. `if (x = 5)` won't compile for ints.
stray ;	`if (x > 5);` or `while (canMove());` ends the statement with an empty body.
== on objects	Checks for the same object (alias), not equal contents. Use `.equals()` for Strings and your own classes.
infinite loop	The while condition never becomes false because nothing in the body changes it.
off by one	`i < n` runs n times; `i <= n` runs n + 1. `i <= s.length()` goes one index too far.
dangling else	Without braces, `else` attaches to the nearest `if`, not the one you indented it under.
wrong order	In an else-if chain, a broad condition placed first swallows the specific cases below it.
flag inside loop	Initialize `found = false` *before* the loop, not inside it, or it resets every pass.
De Morgan half-done	Flip the operator *and* negate both sides: `!(a && b)` is `!a || !b`, not `!a || b`.
nested count	Inner runs fully per outer pass: multiply for independent bounds, add up each pass when the inner bound depends on the outer variable.

Can you…

- [ ] Write a while loop using a Painter Boolean method
- [ ] Explain the four steps of the problem-solving process
- [ ] Trace an if-else and say which branch runs
- [ ] Rewrite a condition using `!`
- [ ] Tell an alias from a separate object, and say what == returns
- [ ] Override `equals()` in a class you wrote
- [ ] Trace a nested if for three different inputs
- [ ] Fill a truth table for && and `||` from memory
- [ ] Explain short-circuit evaluation with `null`
- [ ] Apply De Morgan's Laws and prove it with a truth table
- [ ] Order an else-if chain correctly
- [ ] Name the parts of a for loop and their execution order
- [ ] Convert a while loop to a for loop
- [ ] Reverse a String and check for a palindrome
- [ ] Write the loop bound for a substring window
- [ ] Trace nested loops with a row-per-outer-pass table
- [ ] Count how many times a statement executes