

Unit 3: Boolean Expressions and if Statements

Unit 3 is about making decisions in code. It covers boolean expressions and relational operators, if and if-else statements, else-if chains, compound conditions built with AND, OR, and NOT, De Morgan's laws for simplifying those conditions, and the difference between `==` and `.equals()` when comparing objects.

AP Computer Science A

Boolean Expressions and if Statements

About 14 minutes to read

How to use this guide

Read it in order the first time because the topics stack. Boolean expressions feed the if statement, if statements grow into if-else and else-if chains, and compound conditions with De Morgan's laws are how real conditions get written and simplified. Exam questions usually show you a short code segment and ask for the output or the value of a boolean expression, so trace each example by hand as you read. After the first read, use the trap boxes and the tables to review the distinctions that exam questions test most often. Finish with the practice questions, then complete the recall check on the last page out loud and note any items you cannot explain yet.

What this unit is worth. Unit 3 is about 15 to 17.5 percent of the AP Computer Science A exam, one of the heaviest units. Conditionals appear in almost every free-response question, and De Morgan's law plus short-circuit evaluation are reliable multiple-choice traps.

3.1 Boolean Expressions

A **boolean expression** is any expression that evaluates to `true` or `false`. The **relational operators** compare two values and produce a boolean result.

OPERATOR	MEANING	EXAMPLE (X = 7)
<code>==</code>	equal to	<code>x == 7</code> is true
<code>!=</code>	not equal to	<code>x != 7</code> is false
<code><</code>	less than	<code>x < 10</code> is true
<code>></code>	greater than	<code>x > 10</code> is false
<code><=</code>	less than or equal to	<code>x <= 7</code> is true
<code>>=</code>	greater than or equal to	<code>x >= 8</code> is false

The two sides of a relational operator must be of compatible types. Comparing an `int` with a `double` is fine, but using `<` on two `String` objects is a compile error. The operators `==` and `!=` work on primitives and on objects, though for objects they compare references rather than contents. That distinction is the whole of topic 3.7.

Trap. A single `=` assigns, a double `==` compares, and the compiler treats them very differently inside a condition. `if (x = 5)` does not compile, because the assignment produces an `int` and a condition must be `boolean`. But `if (ready = true)` does compile: it assigns `true` to `ready` and the condition is always true. When a condition behaves strangely, read `=` as "assign" and `==` as "compare" and check which one the code actually says.

3.2 if Statements and Control Flow

An **if statement** runs its body only when its condition is true. The condition must be a boolean expression. This is **control flow**: the condition decides which statements execute next.

```
int age = 16;
if (age >= 16) {
    System.out.println("You can get a license.");
}
System.out.println("Done.");
```

Because `age >= 16` is true, both lines print. If `age` were 14, only "Done." would print. Curly braces group statements into one body. Without braces, only the single statement right after the `if` is controlled, and everything after it runs no matter what.

Trap. A semicolon immediately after the condition ends the `if` with an empty statement. In `if (x > 10); System.out.println("big");` the word "big" prints for every value of `x`, because the `if` controls nothing and the print statement always runs. The compiler accepts it without complaint, so this one is caught only by reading carefully.

3.3 if-else Statements

An **if-else statement** chooses between exactly two paths. When the condition is true the `if` branch runs, otherwise the `else` branch runs. One of the two always runs, and never both.

```
int score = 72;
if (score >= 90) {
    System.out.println("A");
} else {
    System.out.println("Not an A");
}
```

The `else` never takes its own condition. It simply catches every case the `if` condition missed. That makes if-else the right tool whenever a situation splits into two mutually exclusive outcomes.

Trap. Two separate `if` statements are not the same as one if-else. With `if (x > 0) ... if (x < 10) ...` both bodies can run, since each condition is tested independently. With `if (x > 0) ... else ...` exactly one body runs. When you predict output, check whether the code says `else` or starts a fresh `if`.

3.4 else if Statements

An **else-if chain** handles more than two outcomes. The conditions are tested in order from top to bottom, the first true condition wins, and its body runs. Once a body runs, the rest of the chain is skipped. A final `else` catches everything the earlier conditions missed.

```
int n = 85;
if (n >= 90) {
    System.out.println("A");
} else if (n >= 80) {
    System.out.println("B");
} else if (n >= 70) {
    System.out.println("C");
} else {
    System.out.println("F");
}
```

This prints "B". Notice that `n >= 80` and `n >= 70` are both true for 85, but only the first true branch runs. Order matters: the conditions should go from narrowest to broadest, because a broad condition placed first swallows the cases the later conditions were meant to catch.

Trap. An `else` always binds to the nearest unmatched `if`. In nested code without braces, the indentation can lie about which `if` owns the `else`.

```
if (x > 0)
    if (y > 0)
        System.out.println("both positive");
else
    System.out.println("x is not positive");
```

Despite the indentation, the `else` belongs to the inner `if (y > 0)`, not the outer one. If `x` is negative, the outer condition is false, the whole body is skipped, and nothing prints at all. Braces on every nested `if` remove the ambiguity.

3.5 Compound Boolean Expressions

Conditions can combine simpler boolean expressions with the **logical operators** `!` (NOT), `&&` (AND), and `||` (OR). The result follows the truth tables below, where `a` and `b` are any boolean expressions.

EXPRESSION	VALUE
<code>!a</code>	true when <code>a</code> is false, false when <code>a</code> is true
<code>a && b</code>	true only when both <code>a</code> and <code>b</code> are true
<code>a b</code>	true when at least one of <code>a</code> and <code>b</code> is true

When operators mix, `!` is evaluated first, then `&&`, then `||`. Parentheses override that order and usually make the intent clearer. For example, `!a && b` negates only `a` and then ANDs the result with `b`, which is not the same as `!(a && b)`, where the `!` applies to the whole AND expression.

Java evaluates `&&` and `||` with **short-circuit evaluation**. For `a && b`, if `a` is false the result is already decided, so `b` is never evaluated. For `a || b`, if `a` is true, `b` is never evaluated. This is often used deliberately as a guard.

```
if (s != null && s.length() > 0) {  
    System.out.println("nonempty");  
}
```

If `s` is `null`, the left side is false and `s.length()` never runs, which avoids a `NullPointerException`. If Java evaluated both sides, that line would crash on a null string.

Trap. Exam questions love putting dangerous code on the right side of `&&` or `||`: a division by a variable that could be zero, an array access at an index that could be out of bounds, a method call on something that could be null. Before you predict a crash, evaluate the left side first. If short-circuiting kicks in, the dangerous code on the right never executes.

3.6 Equivalent Boolean Expressions

Two boolean expressions are **equivalent** when they agree for every possible input. **De Morgan's laws** give the two equivalences that come up constantly: negating an AND produces an OR of the negations, and negating an OR produces an AND of the negations.

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

When the `!` moves across the parentheses, two things change: each part gets negated, and the operator flips. Both changes are required. Here is the same idea applied to a realistic condition.

```
!(age >= 13 && age <= 19)
// is equivalent to
age < 13 || age > 19
```

Each relational operator has a negation, which is just the operator flipped to cover the opposite cases.

ORIGINAL	NEGATION
<code>x < 5</code>	<code>x >= 5</code>
<code>x > 5</code>	<code>x <= 5</code>
<code>x <= 5</code>	<code>x > 5</code>
<code>x >= 5</code>	<code>x < 5</code>
<code>x == 5</code>	<code>x != 5</code>
<code>x != 5</code>	<code>x == 5</code>

Trap. The classic De Morgan error is negating each part but leaving the operator alone: turning `!(a && b)` into `!a && !b`. That expression means "neither a nor b," which is the negation of `a || b`, not of `a && b`. Whenever you distribute a `!` across parentheses, flip `&&` to `||` and `||` to `&&` as part of the same step.

3.7 Comparing Objects

For objects, `==` and `.equals()` answer different questions. The operator `==` tests **reference equality**: whether two references point to the very same object in memory. The method `.equals()`, for `String`, tests whether the contents match character for character.

```
String s1 = new String("java");
String s2 = new String("java");
System.out.println(s1 == s2);           // false: two different objects
System.out.println(s1.equals(s2));      // true: same contents
```

The rule for the exam is simple. To compare the contents of two strings, always use `.equals()`. Comparing strings with `==` sometimes appears to work, because Java reuses a single object for identical string literals, but that reuse does not apply to strings built at runtime, such as text read from user input. Code like `if (input == "quit")` fails exactly when it matters.

One more caution: calling `.equals()` on a `null` reference throws a `NullPointerException`. Writing the literal first, as in `"quit".equals(input)`, avoids the crash when `input` is `null`. For objects other than `String`, the default `.equals()` behaves like `==` unless the class overrides it.

Trap. Any question that compares two `String` variables with `==` is testing whether you know it checks references. If the strings were created separately, `==` is false even when the text matches. Reach for `.equals()` for contents, and treat `==` on objects as a reference check.

Confusions That Cost Points

PAIR	HOW TO KEEP THEM STRAIGHT
<code>=</code> vs <code>==</code> in a condition	<code>=</code> assigns and <code>==</code> compares. <code>if (x = 5)</code> does not compile; <code>if (b = true)</code> compiles but is always true. Neither is a comparison.
<code>==</code> vs <code>.equals()</code> on Strings	<code>==</code> asks whether two references point to the same object. <code>.equals()</code> asks whether the contents match. Use <code>.equals()</code> for content comparison.
<code>!(a && b)</code> vs <code>!a && !b</code>	Only De Morgan's form is equivalent: <code>!(a && b)</code> equals <code>!a !b</code> . Negating the parts without flipping the operator gives a different expression.
Short-circuit: right side runs vs skipped	With <code>&&</code> , a false left side skips the right side. With <code> </code> , a true left side skips the right side. Dangerous code on the right may never execute.
if-else vs two separate <code>if</code> s	if-else runs exactly one branch. Two separate <code>if</code> s are independent checks, and both bodies can run.
Dangling <code>else</code>	An <code>else</code> binds to the nearest unmatched <code>if</code> , not to the <code>if</code> its indentation suggests. Use braces to make the pairing explicit.

Practice Questions

Original questions written for this guide in the style of the AP exam. Answers and explanations are on the next page, so complete the questions before checking them.

1. Consider the following declarations.

```
int m = 6;  
int n = 2;
```

Which of the following expressions evaluates to `true`?

1. `m % n == 1`
2. `m / n != 3`
3. `m - n >= 4`
4. `m + n <= 7`

2. Consider the following code segment.

```
int temp = 68;  
if (temp > 70)  
    System.out.print("hot");  
if (temp > 60)  
    System.out.print("warm");
```

What is printed as a result of executing the code segment?

1. hotwarm
2. warm
3. hot
4. Nothing is printed

3. Consider the following code segment.

```
int num = 15;  
if (num % 2 == 0)  
    System.out.print("A");  
else  
    System.out.print("B");  
if (num % 3 == 0)  
    System.out.print("C");
```

What is printed as a result of executing the code segment?

1. A
2. B
3. BC
4. ABC

4. Consider the following code segment.

```
int x = 20;  
if (x > 10)  
    System.out.print("one");  
else if (x > 15)  
    System.out.print("two");  
else  
    System.out.print("three");
```

What is printed as a result of executing the code segment?

1. one
2. onetwo
3. onetwothree
4. three

5. Consider the following code segment.

```
int x = 0;
boolean ok = (x != 0) && (10 / x > 2);
System.out.print(ok);
```

What is printed as a result of executing the code segment?

1. true
2. false
3. An ArithmeticException is thrown
4. The code does not compile

6. Which of the following expressions is equivalent to `!(x < 0 || y > 100)` ?

1. `x >= 0 && y <= 100`
2. `x >= 0 || y <= 100`
3. `x > 0 && y < 100`
4. `x < 0 && y > 100`

7. Consider the following code segment.

```
String s1 = new String("java");
String s2 = new String("java");
System.out.print(s1 == s2);
System.out.print(" ");
System.out.print(s1.equals(s2));
```

What is printed as a result of executing the code segment?

1. true true
2. false true
3. true false
4. false false

8. Consider the following code segment.

```
int x = -4;
int y = 9;
if (x > 0)
    if (y > 0)
        System.out.print("A");
else
    System.out.print("B");
```

What is printed as a result of executing the code segment?

1. A
2. B
3. Nothing is printed
4. The code does not compile

Answer Key

- 1. C.** `m - n` is 4, and `4 >= 4` is true. A is wrong because `m % n` is 0, not 1, so `0 == 1` is false. B is wrong because `m / n` is exactly 3 in integer division, so `3 != 3` is false. D is wrong because `m + n` is 8, and `8 <= 7` is false.
- 2. B.** These are two independent `if` statements, not an if-else. The first condition, `68 > 70`, is false, so "hot" is skipped. The second condition, `68 > 60`, is true, so "warm" prints. A treats the two statements as if both conditions held. C reports the first branch as though `68 > 70` were true. D misses that the second condition is satisfied.
- 3. C.** `15 % 2` is 1, so the first condition is false and the `else` prints "B". Then the separate `if (num % 3 == 0)` is true, so "C" prints, giving "BC". A and B each report only part of the output, ignoring the second `if`. D assumes the first condition was true, but 15 is not even.
- 4. A.** In an else-if chain the first true condition wins and the rest of the chain is skipped. `20 > 10` is true, so "one" prints and the chain ends, even though `20 > 15` is also true. B and C wrongly assume that every true condition in the chain executes its body. D would require both earlier conditions to be false, which they are not.
- 5. B.** Short-circuit evaluation decides this. `x != 0` is false, so for `&&` the result is already false and the right side `10 / x > 2` is never evaluated. No division happens, so no exception is thrown and `ok` is false. A ignores the short-circuit and evaluates both sides. C assumes the division always runs; it does not. D is wrong because the code is legal Java.
- 6. A.** De Morgan's law requires two changes when the `!` moves across the parentheses: each part is negated (`<` becomes `>=`, `>` becomes `<=`) and `||` flips to `&&`. B makes only the first change and keeps `||`, which is the classic De Morgan error. C uses strict inequalities, which wrongly excludes the boundary values 0 and 100. D drops the negation entirely.
- 7. B.** `s1` and `s2` are two distinct objects, so `==` compares references and gives false. `.equals()` compares contents, and both hold "java", so it gives true. A assumes `==` compares contents. C reverses the two behaviors. D assumes `.equals()` also compares references, which it does not for `String`.
- 8. C.** The `else` binds to the nearest `if`, which is the inner `if (y > 0)`, not the outer one. The outer condition `-4 > 0` is false, so the entire nested statement is skipped and nothing prints. A and B both assume the `else` belongs to the outer `if`, which the indentation suggests but the language rules reject. D is wrong because the code compiles without error.

When you check your answers, note which distinction each miss came from. Make a flashcard for that distinction and drill it spaced out over the next few days instead of rereading the whole section. If you missed one of these questions, the same distinction is worth practicing again in Rycal, where the AP Computer Science A deck has flashcards for it and more practice questions use the same kinds of traps.

One-Page Recall Check

Say each answer out loud before you look back, and mark the ones you cannot finish. Anything you cannot say out loud yet belongs in your flashcard deck. In Rycal, add those items to the AP Computer Science A deck and let spaced review bring them back over the next few days.

- State what a boolean expression evaluates to, and list the six relational operators.
- Explain why the condition of an `if` statement must be a boolean value.
- Predict the output of two sequential `if` statements versus one if-else, and explain the difference.
- Explain what happens when a semicolon follows the condition of an `if`.
- Describe how an else-if chain chooses its branch, and explain why the order of the conditions matters.
- State which `if` an `else` belongs to when `if` statements are nested without braces.
- Write the truth tables for `&&`, `||`, and `!` from memory.
- Explain short-circuit evaluation, and give a case where it prevents an error.
- State both of De Morgan's laws, and use them to simplify `!(a && b)`.
- Negate each relational operator. For example, the negation of `<` is `>=`.
- Explain the difference between `==` and `.equals()` for `String` objects.
- Explain why comparing user input to a string literal with `==` is unreliable.

Where to go next. Turn every missed item above into flashcards and drill them spaced out over several days rather than in one sitting. In Rycal, open the AP Computer Science A deck. The deck covers the terms in this guide, and its practice questions target the same traps named here. If you have a test date, add it in the Test Planner. You can also start your next review with a Brain Dump, then check what you missed against this guide.

Key terms for this unit

Boolean expression, Relational operators (`==`, `!=`, `<`, `>`, `<=`, `>=`), Control flow, if statement, if-else statement, else-if chain, Logical operators (`!`, `&&`, `||`), Truth table, Operator precedence, Short-circuit evaluation, Equivalent boolean expressions, De Morgan's laws, Reference equality, `.equals()`, String comparison, `NullPointerException`, Dangling else.

About this guide. Written for Rycal and aligned to the College Board AP Computer Science A course framework, Unit 3. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.