

Unit 1: Primitive Types

Unit 1 is where Java begins. It covers declaring variables, the three primitive types you will rely on all year, and doing arithmetic the way Java does it. The rules are few but exact, and the exam tests them constantly, especially integer division and casting.

AP Computer Science A

Primitive Types

About 12 minutes to read

How to use this guide

Read it in order the first time because the topics build on each other. Variables give you the types, expressions do arithmetic with those types, compound operators abbreviate the assignments, and casting converts between types. Exam questions usually show a short snippet and ask for its value or its output, so trace each example with a pencil 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 1 is about 2.5 to 5 percent of the AP CSA exam, but its concepts (especially integer division and casting) appear in questions across every other unit. Getting these mechanics automatic pays off everywhere.

1.1 Why Programming? Why Java?

A **program** is a set of instructions a computer follows to do something useful. Java is a general-purpose programming language, and it is the language of the AP CSA exam. Everything in this course, from Unit 1 through Unit 10, is written in Java.

Java is a **compiled** language. You write source code in a text file, and a **compiler** translates it into **bytecode**, a lower-level form the computer can run. That bytecode runs on the **Java Virtual Machine** (JVM), which exists on many kinds of computers. Because programs run on the JVM instead of directly on one machine's hardware, the same Java program runs in different places without being rewritten. This section is background. The exam tests the Java you write, not the story of the language.

1.2 Variables and Data Types

A **variable** is a named storage location that holds one value. Every variable has a **type**, which decides what kind of value it can hold. Unit 1 uses three **primitive types**, the simple built-in types Java provides.

TYPE	HOLDS	EXAMPLE
int	Whole numbers: positive, negative, or zero	int count = 12;
double	Numbers with decimal points	double price = 4.99;
boolean	The values true or false only	boolean done = false;

Declaring a variable announces its name and type. **Initializing** it gives it a first value. You can do both in one line or in two steps.

```
int total;           // declared
total = 0;           // initialized
int sum = total + 5; // declared and initialized at once
```

Trap. Using a local variable before it has been given a value is a compile error. Java does not fill in a starting value for you. The line `int x; System.out.println(x);` will not compile, because `x` was declared but never initialized.

1.3 Expressions and Assignment Statements

An **expression** is anything that produces a value: a literal like 7, a variable like count, or values and variables combined with **operators** like count + 3. An **assignment statement** stores a value in a variable with a single =. The right side is evaluated first, and the result is stored in the variable on the left. The = means "store," not "equals."

Java follows the order of operations you know from math, with one addition. Multiplication, division, and the remainder operator % run before addition and subtraction. Operators at the same level run left to right. Parentheses override everything.

```
int a = 2 + 3 * 4;    // 14, not 20
int b = 10 - 4 / 2;   // 8, not 3
int c = (10 - 4) / 2; // 3
int d = 20 / 4 * 2;   // 10: left to right, 5 * 2
```

When an expression mixes int and double, the int values are converted and the result is a double. When both operands are int, the result is an int, even when the true quotient has a fractional part. This is **integer division**: 7 / 2 is 3, not 3.5, and 1 / 2 is 0, not 0.5. The remainder is discarded, never rounded.

```
double x = 7 + 2.0; // 9.0: one double operand makes it double
int y = 7 / 2;      // 3: both int, so the .5 is discarded
```

Trap. Integer division is the most tested idea in Unit 1. Before you compute any division, check the operand types. If both are int, the answer is an int with the fraction thrown away. Students lose points by reading 7 / 2 as 3.5.

1.4 Compound Assignment Operators

A **compound assignment operator** does arithmetic on a variable and stores the result back in it, in one step. Each one is shorthand for the longer assignment.

```
int n = 10;
n += 5;    // 15, same as n = n + 5
n -= 3;    // 12, same as n = n - 3
n *= 2;    // 24, same as n = n * 2
n /= 4;    // 6,  same as n = n / 4
n %= 4;    // 2,  same as n = n % 4
```

The type rules do not change just because the statement is shorter. In the example above, `n /= 4` still does integer division, so `24 / 4` gives the `int` 6. And `n %= 4` stores the remainder 2. The operator `+=` is also the standard way to add one: `n += 1` increments `n`.

Trap. Compound assignment keeps the variable's type for you, silently. For an `int n`, the statement `n += 2.5`; compiles and stores 7, because Java inserts the cast to `int` itself. The plain version `n = n + 2.5`; is a compile error, since a `double` result cannot be stored in an `int` without a cast. The shorthand is not always a safe rewrite of the long form.

1.5 Casting and Ranges of Variables

Casting converts a value from one type to another. Write the target type in parentheses in front of the value. A cast from `double` to `int` **truncates toward zero**: it chops off the decimal part and never rounds. `(int) 3.99` is 3, and `(int) -3.99` is -3, not -4.

Where you put the cast decides where the conversion happens, and that changes the answer.

```
int a = 9;
int b = 4;
double x = (double) a / b;    // 2.25: a becomes 9.0 first
double y = (double) (a / b);  // 2.0: integer division runs first
```

In the first line the cast turns `a` into 9.0 before the division, so the division is `double` division. In the second line the parentheses force `a / b` to run first as integer division (`9 / 4 = 2`), and the cast only converts the already truncated 2 into 2.0.

Trap. The type of the variable on the left cannot reach back and change how the right side was computed. In `double c = 5 / 3`;, the division runs first with two `ints`, giving 1, and `c` stores 1.0, not 1.666.... If you want the decimal result, cast before dividing: `(double) 5 / 3`.

Ranges of Variables and Overflow

Every type has a range. An `int` holds whole numbers from -2,147,483,648 to 2,147,483,647. A `double` holds a far wider range, including fractional values. If an `int` calculation passes its maximum, Java does not throw an error. The value **wraps around** to the negative end, silently. After `int big = 2147483647;`
`big = big + 1;`, `big` holds -2147483648.

The exam tests the concept, not the exact boundary numbers: arithmetic that overflows an `int` wraps instead of failing. One more boundary to know: `boolean` values cannot be cast to numbers, and numbers cannot be cast to `boolean`. The line `(int) true` is a compile error.

Confusions That Cost Points

PAIR	HOW TO KEEP THEM STRAIGHT
<code>7 / 2</code> vs <code>7 / 2.0</code>	Two <code>ints</code> give the <code>int</code> 3, with the remainder discarded. One <code>double</code> operand gives 3.5. Read the operand types before you divide.
<code>(double) a / b</code> vs <code>(double) (a / b)</code>	The cast converts only what it touches. Cast an operand to get double division. Wrapping the whole integer division converts the already truncated result.
<code>(int) 3.99</code> vs rounding	Casts truncate toward zero. <code>(int) 3.99</code> is 3 and <code>(int) -3.99</code> is -3. A cast never rounds.
<code>%</code> is the remainder	<code>7 % 2</code> is 1, the leftover after division, not the quotient 3. With negatives the sign follows the left operand: <code>-7 % 3</code> is -1.
Declared vs initialized	<code>int x;</code> announces <code>x</code> but gives it no value. Using <code>x</code> before storing something in it is a compile error.
<code>n += 2.5</code> vs <code>n = n + 2.5</code>	For an <code>int n</code> , the compound form compiles and truncates to 7. The plain form is a compile error. Compound assignment keeps the variable's type for you.
<code>int</code> overflow	Past 2,147,483,647 an <code>int</code> wraps to the negative end instead of erroring. No warning, no exception.

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 code segment.

```
int x = 7;  
int y = 2;  
System.out.println(x / y);
```

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

- 1. 3
- 2. 3.5
- 3. 4
- 4. 3.0

2. Consider the following code segment.

```
double result = 5 + 3 / 2 * 2.0;  
System.out.println(result);
```

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

- 1. 7.0
- 2. 8.0
- 3. 9.0
- 4. 7

3. Consider the following code segment.

```
int a = 9;  
int b = 4;  
System.out.println((double) a / b);
```

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

- 1. 2.0
- 2. 2
- 3. 2.25
- 4. 9.0

4. Consider the following code segment.

```
double x = -7.8;  
int y = (int) x;  
System.out.println(y);
```

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

1. -8
2. -7
3. 7
4. -7.0

5. Consider the following code segment.

```
System.out.println(17 % 5);
```

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

- 1. 3
- 2. 2
- 3. 3.4
- 4. 5

6. Consider the following code segment.

```
int n = 12;  
n /= 5;  
n += 2;  
System.out.println(n);
```

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

- 1. 4
- 2. 4.4
- 3. 2
- 4. 14

7. Consider the following code segment.

```
int a = 5;  
int b = 3;  
double c = a / b;  
System.out.println(c);
```

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

- 1. 1.6666666666666667
- 2. 1.0
- 3. 1
- 4. 2.0

8. Consider the following code segment.

```
int a = -7;  
int b = 3;  
System.out.println(a % b);
```

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

- 1. -1
- 2. 1
- 3. -2
- 4. 2

Answer Key

- 1. A.** Both operands are `int`, so `/` does integer division: $7 / 2$ is 3 with a remainder of 1, and the remainder is discarded. B forgets that `int` divided by `int` never produces a decimal part. C rounds the true quotient up instead of truncating. D has the right digits with the wrong type: the result of integer division is the `int` 3, not the `double` 3.0.
- 2. A.** Multiplication and division run before addition, left to right: $3 / 2$ runs first as integer division and gives 1. Then $1 * 2.0$ is 2.0, and $5 + 2.0$ is 7.0. B treats $3 / 2$ as 1.5, skipping the integer division step that happens before the `double` enters the expression. C rounds the integer division up to 2 instead of truncating to 1. D has the right digits but the wrong type: the expression contains the `double` 2.0, so the result is the `double` 7.0.
- 3. C.** The cast applies to `a` before the division, so `a` becomes 9.0 and $9.0 / 4$ is double division: 2.25. A is what `(double)(a / b)` would give: the parentheses force integer division first ($9 / 4 = 2$) and the cast only converts the 2. B is plain integer division with no cast at all. D stops after the cast and never performs the division.
- 4. B.** Casts truncate toward zero, chopping the decimal part without rounding: `(int) -7.8` is -7. A rounds down to -8, which is what rounding would give, not truncation. C drops the negative sign. D has the right digits but the wrong type: casting to `int` produces an `int`, so -7.0 is impossible.
- 5. B.** The `%` operator gives the remainder: 17 divided by 5 is 3 with a remainder of 2. A is the quotient, confusing `%` with `/`. C is the exact decimal quotient. D is the divisor, which plays no role in the remainder.
- 6. A.** `n /= 5` means `n = n / 5`, and $12 / 5$ is integer division: 2. Then `n += 2` makes it 4. B treats $12 / 5$ as 2.4, forgetting integer division. C ran only the first statement. D ran only the second statement, adding 2 to the original 12.
- 7. B.** The expression `a / b` is computed first with both operands `int`, so $5 / 3$ is 1 by integer division. Only then is the 1 converted to `double` for storage: 1.0. A assumes the `double` on the left reaches back and changes how the division runs. C has the right digits but the wrong type: `c` is a `double`, so it holds 1.0. D rounds the integer division up instead of truncating.
- 8. A.** The `%` operator gives the remainder with the sign of the left operand: -7 equals $(-2)(3) + (-1)$, so the remainder is -1. B has the right magnitude with the wrong sign; 1 would be the answer to $7 \% -3$. C is the quotient, confusing `%` with `/`. D is the remainder with the sign flipped, the answer you get from clock arithmetic instead of Java's rule.

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 Primitive Types 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 Primitive Types deck and let spaced review bring them back over the next few days.

- Explain the difference between declaring and initializing a variable, and state what happens if you use an uninitialized local variable.
- Name the three primitive types in this unit and give an example value for each.
- Evaluate $7 / 2$, $7.0 / 2$, and $7 / 2.0$, and explain why the operand types decide each result.
- State the rule for the type of an arithmetic expression with mixed `int` and `double` operands.
- Evaluate `(int) 3.99` and `(int) -3.99`, and state the rule that casts follow.
- Compute $17 \% 5$ and $-7 \% 3$, and explain what the `%` operator gives you.
- Rewrite `n += 5`, `n -= 5`, `n *= 5`, `n /= 5`, and `n %= 5` as plain assignment statements.
- Explain why `double c = 5 / 3;` stores 1.0, and show two ways to get 1.666... instead.
- State the range of `int` and describe what happens when an `int` calculation exceeds it.
- Explain what it means that Java is a compiled language.

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 Primitive Types deck under AP Computer Science A. 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

Primitive type, Variable, Declaration, Initialization, `int`, `double`, `boolean`, Expression, Operator, Operand, Assignment statement, Order of operations, Integer division, Truncation, Cast, Compound assignment operator, Remainder (`%`), Overflow, Compiler, Bytecode, Java Virtual Machine (JVM).

About this guide. Written for Rycal and aligned to the College Board AP Computer Science A course framework, Unit 1. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.