

1.4 Assignment Statements and Input

Self-Study Guide · About 55 minutes · With answer key

By the end you will be able to

- Use the assignment operator to give a variable its first value or a new one, and predict the value it holds afterward.
- Explain why Java works out the whole right side before storing anything, and why changing one variable later does not update another.
- Work out the type of an expression and decide whether its value can be stored in a given variable.
- Explain why a variable must be assigned before it is used, and what the special value null means.
- Describe several forms of input and read text input with a Scanner.

What to remember from earlier

- A variable is a named storage location that holds one value of its data type. ([Topic 1.2: Variables and Data Types](#), [1.2.4 Declaring a Variable](#))
- The primitive types are int, double, and boolean; any other type is a reference type. ([Topic 1.2: Variables and Data Types](#), [1.2.2 Primitive Types and Reference Types](#))
- A declaration gives the type and the name, as in `int score`; ([Topic 1.2: Variables and Data Types](#), [1.2.4 Declaring a Variable](#))
- `print` stays on the same line, and `println` moves to a new line afterward. ([Topic 1.3: Expressions and Output](#), [1.3.2 print Versus println](#))
- Two int values give an int, so `7 / 2` is 3; with at least one double the result is a double. ([Topic 1.3: Expressions and Output](#), [1.3.6 Integer Division](#))
- An exception is a run-time error that stops a running program. ([Topic 1.1: Introduction to Algorithms, Programming, and Compilers](#), [1.1.8 Run-Time Errors and Exceptions](#))

1.4.1 The Assignment Operator

In Topic 1.2 you declared variables: `int tickets`; sets aside a storage location named `tickets` that can hold one `int`. But a declaration only creates the box. It does not put anything in it, and a program that tracks tickets, scores, or prices needs to store values there and change them as things happen. A movie theater's program must record that you bought 3 tickets, then record 8 when your friends join.

The tool for that is a statement that says "put this value into that variable." An **assignment statement** stores a value in a variable. It uses the **assignment operator**, the single equals sign `=`. The variable goes on the left, the value goes on the right, and the statement ends with a semicolon:

```
1 tickets = 3;
```

Read it as "tickets gets 3" or "store 3 in tickets," not as "tickets equals 3." That wording matters. In math, `=` states a fact that is always true. In Java, `=` is an action that happens once, at the moment that line runs.

The first time a variable is assigned a value, the variable is **initialized**. You can initialize a variable on a later line, as above, or in the declaration itself, as in `int lives = 3;`. Any assignment after that **changes** the value: the new value replaces the old one, and the old one is gone. A variable holds exactly one value at a time.

KEY IDEA

The assignment operator `=` stores the value on its right in the variable on its left. The first assignment initializes the variable; each later assignment replaces the value it held.

WORKED EXAMPLE Initializing, then changing

Watch what `tickets` holds after each line.

```
1 int tickets;
2 tickets = 3;
3 System.out.println(tickets);
4 tickets = 8;
5 System.out.println(tickets);
```

Line	What happens	tickets holds	Printed
1	Declared; no value yet	(nothing)	
2	Initialized: 3 is stored	3	
3	The current value is printed	3	3
4	Changed: 8 replaces 3, and 3 is gone for good	8	
5	The current value is printed	8	8

```
3
8
```

Line 3 and line 5 are the same statement, yet they print different things, because each one prints whatever `tickets` holds at the moment that line runs.

Now change one thing: initialize the variable in its declaration instead of on its own line.

```
1 int lives = 3;
2 System.out.println(lives);
3 lives = 2;
4 System.out.println(lives);
```

```
3
2
```

Line 1 declares `lives` and initializes it to 3 in one statement. Line 3 is an ordinary assignment with no type in front, because `lives` already exists. Writing the type only belongs in the declaration; the type tells Java to create a new variable, and you create each variable once.

WATCH OUT

Writing the assignment backward. Because `=` looks like the equals sign from math, where `x = 10` and `10 = x` mean the same thing, students sometimes put the value on the left:

```
1 int score = 0;
2 10 = score;
```

This does not compile. Java reports:

```
error: unexpected type
    required: variable
    found:    value
```

The left side of `=` must be a variable, because it names the place where the value will be stored. `10` is a value, not a storage location, so there is nowhere to put anything. (The exact wording can vary between Java versions.) The fix is to put the variable on the left:

```
score = 10;.
```

PAUSE AND TRY**Question 1** Multiple choice

What does this code print?

```
1 int laps = 2;
2 laps = 6;
3 laps = 4;
4 System.out.println(laps);
```

- A. 2
- B. 4
- C. 6
- D. 12

Question 2 Debugging

This code should store 100 in `highScore` and print it, but it does not compile. Explain the error and write the corrected line.

```
1 int highScore = 0;
2 100 = highScore;
3 System.out.println(highScore);
```

1.4.2 The Right Side Is Evaluated First

So far the right side of every assignment has been a single number. But programs usually need to compute the new value, often from the variable's own current value. A game adds 5 points to a score; a store multiplies a price by a quantity. In Topic 1.3 you wrote arithmetic expressions such as `4 * 3` and printed their results. Now you can store those results.

An **expression** is any piece of code that produces a value: a literal such as `5`, a variable such as `points`, or a combination of them with operators, such as `points + 5`. When an expression is evaluated, it produces **one single value**. That is exactly what an assignment needs.

Java carries out every assignment in two steps, always in this order:

1. **Evaluate** the whole expression on the right, using the values the variables hold right now, to get one value.
2. **Store** that value in the variable on the left.

Because the right side is finished before anything is stored, a variable can appear on both sides. The right side uses the old value, and only then does the new value replace it.

KEY IDEA

Java evaluates the entire right side to one value first, using the current values of any variables in it, and only then stores that value in the variable on the left.

WORKED EXAMPLE Using a variable's own value

```
1 int points = 10;
2 points = points + 5;
3 System.out.println(points);
```

Line 2 looks impossible as math (no number equals itself plus 5), but as an action it is ordinary. Here are the two steps:

Step	Work	points holds
Before line 2		10
Evaluate right side	<code>points + 5</code> → <code>10 + 5</code> → 15	10 (still)
Store on the left	15 replaces 10	15

15

During the evaluate step, `points` still holds 10. It changes only in the store step.

Now vary the right side so it is a different variable. When the expression is just a variable, its value is **copied** into the variable on the left. After that, the two variables are separate boxes that happen to hold the same number.

```
1 int monday = 12;
2 int tuesday = monday;
3 monday = 20;
4 System.out.println(monday);
5 System.out.println(tuesday);
```

Line	Work	monday	tuesday
1	Store 12	12	
2	Evaluate monday → 12; store it	12	12
3	Store 20 in monday	20	12

20
12

Line 3 changes only `monday`. Line 2 copied the value 12 into `tuesday`; it did not link the two variables, so `tuesday` keeps 12.

WATCH OUT

Expecting an assignment to update itself later. In a spreadsheet, a cell that holds a formula updates whenever the cells it uses change. Java assignments do not work that way, and students who expect them to are surprised by code like this:

```
1 int price = 4;
2 int quantity = 3;
3 int total = price * quantity;
4 quantity = 5;
5 System.out.println(total);
```

12

You might expect 20, since 4×5 is 20. But line 3 ran once, when `quantity` was 3, and stored the single value 12. `total` holds a number, not a formula. Changing `quantity` on line 4 does nothing to `total`. Java gives no error here, because nothing is illegal: this is a logic error (Topic 1.1), where the program runs but gives the wrong answer.

The fix is to run the assignment again after the change:

```
1 int price = 4;
2 int quantity = 3;
3 int total = price * quantity;
4 quantity = 5;
5 total = price * quantity;
6 System.out.println(total);
```

20

PAUSE AND TRY

Question 3 Tracing

Write the exact output. Use a trace table with columns for `home` and `away`.

```
1 int home = 5;
2 int away = home;
3 home = home + 3;
4 away = away * 2;
5 System.out.println(home);
6 System.out.println(away);
```

Question 4 Multiple choice

What does this code print?

```
1 int width = 4;
2 int area = width * 3;
3 width = 10;
4 System.out.println(area);
```

- A. 12
- B. 30
- C. 10
- D. Nothing; it does not compile

1.4.3 Every Expression Has a Type

Every variable has a type, chosen when it is declared. Every value has a type too. So when an assignment stores the value of an expression, Java has to know: is that value an `int`, a `double`, or a `boolean`? And does it fit the variable?

The answer is that the value of an expression has a type, and that type comes from how the expression is evaluated. You already know the rules from Topic 1.3. An operation on two `int` values produces an `int`; an operation with at least one `double` produces a `double`. Java applies those rules one operation at a time, following the order of operations, and the type of the last operation's result is the type of the whole expression.

The value you store must come from a **compatible** type: one the variable's type accepts. An `int` variable accepts `int` values, a `double` variable accepts `double` values, and a `boolean` variable accepts `true` or `false`. (Topic 1.5 shows a few cases where Java converts a value from one type to another for you.) Why is Java strict about this? Because a variable's type decides how much memory it uses and what it can hold. An `int` variable has no place to keep the `.33` of `84.33`, so Java will not silently throw it away.

KEY IDEA

Evaluating an expression produces one value, and that value has a type determined by the operations that produced it. You can store it only in a variable whose type is compatible.

WORKED EXAMPLE Averaging quiz scores

Three quiz scores are 85, 90, and 78. Look at the type of each right side.

```

1 int total = 85 + 90 + 78;
2 double average = total / 3.0;
3 int roughAverage = total / 3;
4 System.out.println(total);
5 System.out.println(average);
6 System.out.println(roughAverage);

```

Line	Evaluate the right side	Type	Stored
1	85 + 90 → 175, then 175 + 78 → 253	int	253
2	total / 3.0 → 253 / 3.0: one double, so a double	double	84.33333333333333
3	total / 3 → 253 / 3: two ints, so integer division	int	84

```

253
84.33333333333333
84

```

Line 2 divides by 3.0, a double, so the result is a double and fits in the double variable average. Line 3 divides two int values, so the result is the int 84 (the decimal part is dropped, as in Topic 1.3), which fits in the int variable roughAverage. In both lines the type of the expression matches the type of the variable.

The type comes from the operations, not from how the answer looks. `10 / 4 * 2.0` ends up as 4.0, a double, even though 4 is a whole number: `10 / 4` is the int 2, and then `2 * 2.0` involves a double, so the result is the double 4.0.

WATCH OUT

Storing a double result in an int variable. Students often declare a variable as int out of habit, then divide by a double:

```

1 int total = 253;
2 int average = total / 3.0;

```

The right side, `253 / 3.0`, is a double, but average is an int. This does not compile. Java reports something like:

```
error: incompatible types: possible lossy conversion from double to int
```

"Lossy" means the decimal part would be lost. The compiler refuses because it cannot know whether you meant to lose it. The fix is to declare the variable with a type that matches the value: `double average = total / 3.0;`. The same error family appears for other mismatches; for example, `int count = true;` gives
incompatible types: boolean cannot be converted to int.

PAUSE AND TRY**Question 5** Multiple choice

What are the type and the value of the expression `9 / 2 + 0.5`?

- A. `double` value `5.0`
- B. `double` value `4.5`
- C. `int` value `4`
- D. `int` value `5`

Question 6 Debugging

This code should store half of 9 and print `4.5`, but it does not compile. Explain why and write the corrected line.

```
1 int half = 9 / 2.0;
2 System.out.println(half);
```

1.4.4 Assign Before You Use

A declaration creates a variable with no value in it. What should happen if a program tries to use that variable before anything has been stored there? Back in 1.4.1, the trace table showed `(nothing)` for a variable that was only declared. An expression cannot compute `50 + nothing`.

Java's rule: every variable must be assigned a value before it is used in an expression. The compiler checks this before the program ever runs. It looks at each place a variable is used and asks: on every path to this line, has the variable already been assigned? If not, it refuses to compile.

Why not just use 0? Because 0 might be wrong. A bonus of 0, a price of 0, a temperature of 0: each is a real value that would quietly produce a believable but wrong answer, a logic error that is hard to find. A compile error forces you to decide what the starting value should be.

KEY IDEA

A variable must be initialized before it is used in an expression. Declaring it is not enough.

WORKED EXAMPLE Declared now, initialized later

```
1 int steps;
2 steps = 4000;
3 int goal = steps + 1000;
4 System.out.println(goal);
```

Line	What happens	steps	goal
1	steps is declared with no value	(nothing)	
2	steps is initialized	4000	
3	steps + 1000 → 4000 + 1000 → 5000	4000	5000

```
5000
```

This compiles because `steps` is initialized on line 2, before line 3 uses it. Declaring and initializing on separate lines is allowed; the order is what matters.

Now remove the assignment and see what changes:

```
1 int bonus;  
2 int total = 50 + bonus;  
3 System.out.println(total);
```

This does not compile. Java reports something like:

```
error: variable bonus might not have been initialized
```

Line 2 uses `bonus` in the expression `50 + bonus`, but no line before it ever stored a value in `bonus`. The fix is to initialize it first, for example `int bonus = 0;`, if 0 really is the bonus you mean.

WATCH OUT

Thinking a declaration gives a default value. Students who have seen other languages, or who picture an empty box as "zero," expect `int bonus;` to hold 0. It holds nothing usable, and the compiler says so with `might not have been initialized`. The error points to the line where the variable is *used*, not where it was declared, so look upward from that line for the missing assignment.

PAUSE AND TRY

Question 7 Multiple choice

Which code segment does **not** compile?

- A. `int a; then a = 3; then int b = a + 1;`
- B. `int e; then int f = e + 2;`
- C. `int c = 0; then c = c + 1;`
- D. `int d; then d = 5;`

Question 8 Short answer

Java could have been designed to give every declared variable a starting value of 0 automatically. Explain why refusing to compile is more helpful to a programmer who forgot to initialize a variable.

1.4.5 Reference Variables and null

Everything so far has used primitive types: `int`, `double`, and `boolean`. Recall from Topic 1.2 that any other type is a reference type, and that reference types are used for objects, bundles of related information such as a piece of text. `String`, the type for text such as "Ada", is a reference type. A variable of a reference type does not hold a number or `true/false`. It is connected to an

object. Topics 1.12 and 1.13 explain objects and that connection in detail. For this chapter, two facts are enough: a reference variable can be assigned an object (you have already seen string literals, and in 1.4.6 you will create a new object for reading input), or it can be assigned the special value `null`.

Why would you need a special value? Sometimes a program needs a reference variable before it has anything to connect it to. A sign-up form asks for a nickname, but a player may not have one yet. The program must still initialize the variable (1.4.4), and it should say clearly "no nickname yet" rather than invent one.

The literal `null` is a special value meaning that a reference variable is not connected to any object. Like `3` or `true`, it is a literal: a fixed value written directly in the code. It is not text, so it has no double quotes.

KEY IDEA

A reference variable can be assigned an object or `null`. The literal `null` means the variable is not associated with any object.

WORKED EXAMPLE A nickname that arrives later

```
1 String nickname = null;
2 System.out.println(nickname);
3 nickname = "Ace";
4 System.out.println(nickname);
```

Line	What happens	nickname holds	Printed
1	Initialized to <code>null</code> : no object yet	<code>null</code>	
2	<code>println</code> displays the value <code>null</code>	<code>null</code>	<code>null</code>
3	Assigned the string <code>"Ace"</code> , replacing <code>null</code>	connected to <code>"Ace"</code>	
4	<code>println</code> displays the text	connected to <code>"Ace"</code>	<code>Ace</code>

```
null
Ace
```

Line 1 counts as initialization, so line 2 compiles. Printing a variable that holds `null` displays the word `null`. Line 3 is an ordinary assignment: the new value replaces the old one, just as with an `int`.

Now vary the type. `null` belongs only to reference types. A primitive variable always holds an actual number or `true/false`, so "no object" makes no sense for it:

```
1 int age = null;
```

This does not compile. Java reports something like:

```
error: incompatible types: <null> cannot be converted to int
```

This is the compatible-type rule from 1.4.3 again: `null` is not a value an `int` can hold. If you need "no age entered yet" for an `int`, pick a real starting value, such as 0, and be clear about what it means.

WATCH OUT

Confusing `null` with `"null"`. `String nickname = null;` and `String nickname = "null";` both print `null`, but they are different. The first has no quotes: the variable is connected to no object. The second has quotes: it is a string literal made of the four letters n, u, l, l, a real piece of text that just happens to spell that word. Students mix them up because the printed output looks identical. Write `null` without quotes when you mean "no object."

PAUSE AND TRY

Question 9 Tracing

Write the exact output.

```
1 String team = null;
2 System.out.println(team);
3 team = "Hawks";
4 System.out.println(team);
5 team = null;
6 System.out.println(team);
```

Question 10 Multiple choice

Which declaration compiles?

- A. `int score = null;`
- B. `double price = null;`
- C. `String city = null;`
- D. `boolean done = null;`

1.4.6 Input and the Scanner Class

Every program so far has had its values written into the code. The quiz scores, the ticket counts, the nickname: to change any of them, you would have to edit the program and run it again. Real programs take in information while they run. **Input** is data a program receives from outside itself. Input comes in many forms:

Form	What the program receives	Everyday example
Tactile	Touches, presses, or movements	Tapping a phone screen, a game button
Audio	Sound	Speaking to a voice assistant
Visual	Images or video	A camera scanning a QR code
Text	Characters	Typing your age into a form

This course works with text input. In Java, one way to get text that the user types on the keyboard is the `Scanner` class: a ready-made part of Java whose job is reading text and turning it into values

your program can store. (Exactly what kind of input a particular program accepts is up to its designer, and the AP exam does not test any particular one; you will use `Scanner` in class programs and again in Topics 1.13 and 1.14.)

Here is a complete program that asks for the user's age and stores it:

```

1  import java.util.Scanner;
2
3  public class AgeFromKeyboard
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner(System.in);
8          System.out.print("How old are you? ");
9          int age = input.nextInt();
10         System.out.print("Age: ");
11         System.out.println(age);
12     }
13 }

```

Take it apart one new piece at a time:

Code	What it does
<code>import java.util.Scanner;</code>	Placed at the very top, before the class. Tells Java where to find <code>Scanner</code> , which is not automatically available to every program.
<code>Scanner input</code>	Declares a variable named <code>input</code> of type <code>Scanner</code> , a reference type.
<code>= new Scanner(System.in);</code>	Creates a new <code>Scanner</code> object and assigns it to <code>input</code> . <code>System.in</code> stands for the keyboard.
<code>int age = input.nextInt();</code>	Waits for the user to type a whole number and press Enter, then stores that number in <code>age</code> .

Two pieces use ideas taught later. `new Scanner(System.in)` creates an object, which Topic 1.13 explains fully; for now, use this line as a pattern whenever you need keyboard input.

`input.nextInt()` asks the `Scanner` named `input` for its next whole number; Topic 1.14 explains this kind of call, so for now read it as "ask `input` for the next whole number." Notice that

`int age = input.nextInt();` is still an ordinary assignment from 1.4.2: the right side produces one `int` value, and it is stored in `age`.

A program that waits for typing gives different output every time it runs, depending on what someone types. To make examples you can trace exactly, this guide gives the `Scanner` its input as a string instead of the keyboard: `new Scanner("16")` reads from the text `16` exactly as if a user had typed it. Everything else, including `nextInt()`, works the same way.

KEY IDEA

A `Scanner` reads text input. Import it with `import java.util.Scanner;` then create one with `new Scanner(System.in)` to read the keyboard, or with `new Scanner` and a string to read that text instead.

WORKED EXAMPLE Reading an age from text

AgeFromText.java

```

1  import java.util.Scanner;
2
3  public class AgeFromText
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("16");
8          int age = input.nextInt();
9          System.out.print("Age: ");
10         System.out.println(age);
11     }
12 }

```

Line	What happens	age
1	Makes Scanner available to this program	
7	Creates a Scanner that reads the text 16; stores it in input	
8	input.nextInt() reads 16 and produces the int 16; it is stored	16
9–10	Prints the label, then the value on the same line	16

Age: 16

Swap "16" for `System.in` on line 7 and the program behaves like `AgeFromKeyboard` above: it reads whatever number the user types instead.

WATCH OUT

Forgetting the import. `System` needs no import, so students expect the same of `Scanner`:

NoImport.java

```

1  public class NoImport
2  {
3      public static void main(String[] args)
4      {
5          Scanner input = new Scanner("16");
6          int age = input.nextInt();
7          System.out.println(age);
8      }
9  }

```

This does not compile. Java does not know what `Scanner` is, and reports (once for each place the name appears on line 5):

```

error: cannot find symbol
symbol:   class Scanner

```

The fix is to add `import java.util.Scanner;` as the first line of the file, before `public class`. Topic 1.7 explains where classes such as `Scanner` live and why some need an import.

PAUSE AND TRY**Question 11** Multiple choice

A phone app lets you unlock it by looking at the camera, by saying a passphrase, by pressing your thumb on a sensor, or by typing a PIN. Which of these is the kind of input a `Scanner` reads?

- A. Looking at the camera
- B. Saying a passphrase
- C. Pressing your thumb on the sensor
- D. Typing a PIN

Question 12 Short answer

This guide's examples create a `Scanner` with `new Scanner("16")` instead of `new Scanner(System.in)`. Explain what is different about where each one gets its input, and why the guide uses the string version for examples.

1.4.7 Reading Values with a Scanner

Most programs need more than one input, and not all of them are whole numbers. An order form needs a customer's name (text), a quantity (a whole number), and a price (a decimal). A `Scanner` has a different reading call for each type, and each call produces one value you can assign to a variable of that type.

Picture the input as a line of text with a marker showing where the `Scanner` will read next. Every read takes the next piece from the marker onward, and the marker moves past it. Pieces are separated by spaces or line breaks.

Call	Reads	Produces a
<code>nextInt()</code>	The next piece, as a whole number	<code>int</code>
<code>nextDouble()</code>	The next piece, as a number that may have a decimal point	<code>double</code>
<code>nextLine()</code>	Everything up to the end of the current line, spaces included	<code>String</code>

`nextInt()` and `nextDouble()` skip over any spaces or line breaks before their piece. `nextLine()` returns the rest of the line without the line break itself, and moves the marker to the start of the next line.

KEY IDEA

Each reading call takes the next piece of input and produces one value of its type. Calls read in order, and each result is stored with an ordinary assignment.

WORKED EXAMPLE Reading an order

The input text has a name on the first line and two numbers on the second. In a string literal, `\n` is a line break (Topic 1.3), so this is the same as a user typing `Ada Lovelace`, pressing Enter, then typing `3 2.5`.

OrderReader.java

```

1  import java.util.Scanner;
2
3  public class OrderReader
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("Ada Lovelace\n3 2.5");
8          String customer = input.nextLine();
9          int quantity = input.nextInt();
10         double price = input.nextDouble();
11         double cost = quantity * price;
12         System.out.println(customer);
13         System.out.print("Cost: ");
14         System.out.println(cost);
15     }
16 }

```

The ▶ shows where input will read next.

Line	Call / work	Value stored	Input left after the line
7	(Scanner created)		▶Ada Lovelace / 3 2.5
8	input.nextLine()	customer ← "Ada Lovelace"	▶3 2.5
9	input.nextInt()	quantity ← 3	▶ 2.5
10	input.nextDouble()	price ← 2.5	(nothing left)
11	3 * 2.5 → 7.5	cost ← 7.5	

```

Ada Lovelace
Cost: 7.5

```

Line 8 reads the whole first line, including the space between the two names, because `nextLine` stops only at the line break. Line 11 is the type rule from 1.4.3: an `int` times a `double` is a `double`, so `cost` must be a `double`.

Now change only the input text, spreading the numbers over extra lines and spaces:

OrderReaderLines.java

```
1 import java.util.Scanner;
2
3 public class OrderReaderLines
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("Ada Lovelace\n3\n\n 2.5");
8         String customer = input.nextLine();
9         int quantity = input.nextInt();
10        double price = input.nextDouble();
11        double cost = quantity * price;
12        System.out.println(customer);
13        System.out.print("Cost: ");
14        System.out.println(cost);
15    }
16 }
```

```
Ada Lovelace
Cost: 7.5
```

The output is identical. `nextInt()` and `nextDouble()` skip the line breaks and spaces to find their pieces, so what matters is the **order** of the pieces, not how they are spaced. The order of the reading calls must match the order of the input.

WATCH OUT

Input that does not match the call. `nextInt()` can only produce a whole number. If the next piece is something else, such as a word, the program crashes while it runs:

WrongInput.java

```
1 import java.util.Scanner;
2
3 public class WrongInput
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("three 2.5");
8         System.out.println("Reading the quantity...");
9         int quantity = input.nextInt();
10        System.out.println(quantity);
11    }
12 }
```

```
Reading the quantity...
Exception in thread "main" java.util.InputMismatchException
```

The program compiles, because nothing in the code is wrong as Java syntax. The problem appears only at run time, on line 9, when the next piece turns out to be `three`. An **InputMismatchException** is the exception (Topic 1.1) a `Scanner` throws when the input does not match the type you asked for. Line 8 already ran, so its message appears before the

crash; line 10 never runs. (The exception message also lists lines that show where it happened; their details vary between Java versions.) The fix is input that matches: with "3 2.5" the same program reads 3. When a real user types the input, they must type what the program asks for, which is why programs print a prompt such as `How old are you?` first.

PAUSE AND TRY

Question 13 Tracing

Write the exact output. Track what each variable holds.

ReadsInOrder.java

```
1 import java.util.Scanner;
2
3 public class ReadsInOrder
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("8 2.5 6");
8         int first = input.nextInt();
9         double second = input.nextDouble();
10        int third = input.nextInt();
11        int sum = first + third;
12        System.out.println(sum);
13        System.out.println(second);
14    }
15 }
```

Question 14 Multiple choice

A `Scanner` is created with `new Scanner("Blue Team\n7")`. The first reading call is `String team = input.nextLine();`. What does `team` hold?

- A. "Blue"
- B. "Blue Team"
- C. "Blue Team 7"
- D. "7"

Putting it together

This program estimates the fuel cost of a trip. It reads the trip's name, the distance in miles, the car's miles per gallon, and the price of one gallon, then computes the cost.

TripCost.java

```

1  import java.util.Scanner;
2
3  public class TripCost
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("Beach day\n240 30\n3.5");
8          String tripName = input.nextLine();
9          int miles = input.nextInt();
10         int milesPerGallon = input.nextInt();
11         double pricePerGallon = input.nextDouble();
12         int gallons = miles / milesPerGallon;
13         double cost = gallons * pricePerGallon;
14         System.out.println(tripName);
15         System.out.print("Gallons: ");
16         System.out.println(gallons);
17         System.out.print("Cost: ");
18         System.out.println(cost);
19         miles = miles + 60;
20         System.out.print("Gallons still stored: ");
21         System.out.println(gallons);
22     }
23 }

```

First, the reads (the ▶ marks where `input` will read next):

Line	Call	Value stored	Input left after the line
7	(Scanner created)		▶Beach day / 240 30 / 3.5
8	<code>input.nextLine()</code>	<code>tripName</code> ← "Beach day"	▶240 30 / 3.5
9	<code>input.nextInt()</code>	<code>miles</code> ← 240	▶ 30 / 3.5
10	<code>input.nextInt()</code>	<code>milesPerGallon</code> ← 30	▶ / 3.5
11	<code>input.nextDouble()</code>	<code>pricePerGallon</code> ← 3.5	(nothing left)

Then the computations. For each right side, evaluate first, check the type, then store:

Line	Evaluate the right side	Type	Stored
12	$240 / 30 \rightarrow 8$ (two int values)	int	<code>gallons</code> ← 8
13	$8 * 3.5 \rightarrow 28.0$ (one double)	double	<code>cost</code> ← 28.0
19	$miles + 60 \rightarrow 240 + 60 \rightarrow 300$	int	<code>miles</code> ← 300

Lines 14 to 18 print the name and the two results. Line 19 changes `miles`, but line 12 ran long before and stored the single value 8 in `gallons`, so lines 20 and 21 still print 8:

```

Beach day
Gallons: 8
Cost: 28.0
Gallons still stored: 8

```

Every idea from the chapter appears. The `Scanner` is a new object assigned to a reference variable and needs the import on line 1. Each reading call produces one value of its type, stored with an

ordinary assignment, and the calls read in order. Each right side is evaluated to one value whose type must be compatible with its variable: `cost` must be a `double` because `gallons * pricePerGallon` is a `double`. Every variable is initialized before it is used. And an assignment stores a value, not a formula, so changing `miles` does not change `gallons`.

Common mistakes

The compiler messages from this chapter, collected for quick lookup. Wording can vary slightly between Java versions.

Java says	Usual cause	Fix
unexpected type (required: variable, found: value)	Value on the left of <code>=</code> , as in <code>10 = score;</code>	Put the variable on the left: <code>score = 10;</code>
incompatible types: possible lossy conversion from double to int	A <code>double</code> expression stored in an <code>int</code> variable	Declare the variable as <code>double</code>
incompatible types: <code><null></code> cannot be converted to int	<code>null</code> stored in a primitive variable	Use <code>null</code> only with reference types like <code>String</code>
variable ... might not have been initialized	Variable used before any value was stored	Assign a value before the line that uses it
cannot find symbol (symbol: class Scanner)	Missing import	Add <code>import java.util.Scanner;</code> at the top
<code>InputMismatchException</code> (while running)	Input piece does not match the reading call	Read in the order the input appears; match the types

Vocabulary

Term	Definition	Example
assignment statement	A statement that stores a value in a variable	<code>tickets = 8;</code>
assignment operator	The <code>=</code> sign; stores the value on its right in the variable on its left	<code>lives = 2;</code>
initialized	Describes a variable that has been assigned its first value	<code>int lives = 3;</code>
expression	Code that is evaluated to produce one value, which has a type	<code>total / 3.0</code> (a <code>double</code>)
compatible	Describes a value whose type the variable's type accepts	an <code>int</code> value for an <code>int</code> variable
null	A literal meaning a reference variable is not connected to any object	<code>String nickname = null;</code>
input	Data a program receives from outside itself: tactile, audio, visual, or text	a typed age

Scanner	A Java class that reads text input and turns it into values	<code>new Scanner(System.in)</code>
InputMismatchException	The exception thrown when input does not match the type a <code>Scanner</code> call asks for	<code>nextInt()</code> reading three

Chapter summary

- An assignment statement uses `=` to store the value on its right in the variable on its left. The first assignment initializes the variable; later ones replace its value.
- Java evaluates the whole right side to one value first, using the variables' current values, then stores it. The variable holds that value, not a formula, so later changes to other variables do not update it.
- Every expression's value has a type that comes from its operations. That value can be stored only in a variable of a compatible type, and a variable must be initialized before it is used in an expression.
- A reference variable, such as a `String`, can be assigned an object or `null`, which means "no object." Primitive variables cannot hold `null`.
- Input can be tactile, audio, visual, or text. A `Scanner` (imported from `java.util`) reads text, from the keyboard with `System.in` or from a string, and its `nextInt`, `nextDouble`, and `nextLine` calls read the input in order.

Review questions

Question 15 Multiple choice

What does this code print?

```
1 double result = 11 / 4 * 2.0 + 1;
2 System.out.println(result);
```

- A. 6.5
- B. 5.0
- C. 5
- D. Nothing; it does not compile

Question 16 Tracing

A student wants to swap the values of `first` and `second`, then add 1 to `first`. Write the exact output, and say whether the swap worked.

```
1 int first = 5;
2 int second = 9;
3 first = second;
4 second = first;
5 first = first + 1;
6 System.out.println(first);
7 System.out.println(second);
```

Question 17 Multiple choice

Which declaration does **not** compile?

- A. `String coach = null;`
- B. `double rate = 7 / 2.0;`
- C. `int laps = 12 / 5;`
- D. `int total = 3.0 * 4;`

Question 18 Short answer

Here are two versions of the same code. Version 1 does not compile. Version 2 prints `null`. Explain the difference.

Version 1:

```
1 String motto;  
2 System.out.println(motto);
```

Version 2:

```
1 String motto = null;  
2 System.out.println(motto);
```

Question 19 Debugging

This program should read a quiz name and a score of 17 out of 20, start with no bonus points, and print the percentage 85.0. It has **two** compile errors. Find both, explain each, and write the corrected lines.

QuizScores.java

```
1 import java.util.Scanner;  
2  
3 public class QuizScores  
4 {  
5     public static void main(String[] args)  
6     {  
7         Scanner input = new Scanner("Unit 1 quiz\n17 20");  
8         String quizName = input.nextLine();  
9         int myScore = input.nextInt();  
10        int maxPoints = input.nextInt();  
11        int bonus = null;  
12        int percent = (myScore + bonus) * 100.0 / maxPoints;  
13        System.out.println(quizName);  
14        System.out.println(percent);  
15    }  
16 }
```

Question 20 Tracing

Fourteen boxes are shared among four helpers. Write the exact output.

MovingBoxes.java

```
1 import java.util.Scanner;
2
3 public class MovingBoxes
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("Maple Street\n14 4 2.5");
8         String street = input.nextLine();
9         int boxes = input.nextInt();
10        int helpers = input.nextInt();
11        double poundsPerBox = input.nextDouble();
12        int boxesEach = boxes / helpers;
13        int leftOver = boxes % helpers;
14        double totalPounds = boxes * poundsPerBox;
15        System.out.println(street);
16        System.out.println(boxesEach);
17        System.out.println(leftOver);
18        System.out.println(totalPounds);
19    }
20 }
```

Question 21 Multiple choice

A program reads the hours someone worked:

HalfInput.java

```
1 import java.util.Scanner;
2
3 public class HalfInput
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = new Scanner("4.5");
8         int hours = input.nextInt();
9         System.out.println(hours);
10    }
11 }
```

What happens when it runs?

- A. It prints 4.
- B. It prints 4.5.
- C. It does not compile, because 4.5 is not an int.
- D. It compiles, then stops with an InputMismatchException on line 8.

Question 22 Multiple choice

Which statement about input and Scanner is true?

- A. A Scanner is the only way a program can receive input.
- B. new Scanner(System.in) reads the keyboard, new Scanner("12 7") reads that text, and reading calls such as nextInt() work the same way for both.
- C. The line import java.util.Scanner; is optional, because every class in Java is available to every program.

D. A `Scanner` reads all of the input into its variables the moment it is created.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can use `=` to initialize a variable or change its value, and say what it holds afterward. Reread: 1.4.1 The Assignment Operator
- ☐ I can explain why the right side is evaluated before anything is stored, and why an assignment is not a formula. Reread: 1.4.2 The Right Side Is Evaluated First
- ☐ I can find the type of an expression and decide whether it can be stored in a variable. Reread: 1.4.3 Every Expression Has a Type
- ☐ I can explain why a variable must be initialized before it is used in an expression. Reread: 1.4.4 Assign Before You Use
- ☐ I can explain what null means and which variables can hold it. Reread: 1.4.5 Reference Variables and null
- ☐ I can name forms of input and set up a `Scanner`, including its import line. Reread: 1.4.6 Input and the `Scanner` Class
- ☐ I can trace `nextInt`, `nextDouble`, and `nextLine` reading input in order. Reread: 1.4.7 Reading Values with a `Scanner`

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. Each assignment replaces the value before it. `laps` is initialized to 2, then 6 replaces 2, then 4 replaces 6, so line 4 prints 4.

- **A** is wrong: 2 was only the starting value. Initializing a variable does not lock it.
- **C** is wrong: 6 was replaced by the last assignment. A variable remembers only its most recent value.
- **D** is wrong: assignment never adds values together. $2 + 6 + 4 = 12$ treats `=` as if it accumulated values.

Reread 1.4.1 *The Assignment Operator* if you chose differently.

Question 2

Corrected line 2: `highScore = 100;`

The left side of `=` must be the variable that receives the value. Line 2 has the value 100 on the left, which is not a storage location, so the compiler reports `unexpected type` (it needed a variable and found a value). With the variable on the left, the code compiles and prints 100.

Reread 1.4.1 *The Assignment Operator* if you missed the error.

Question 3

8
10

Line	Work	home	away
1	Store 5	5	
2	Copy the value of home	5	5
3	<code>home + 3</code> → <code>5 + 3</code> → 8; store it	8	5
4	<code>away * 2</code> → <code>5 * 2</code> → 10; store it	8	10

Line 2 copied 5 into `away`, so line 3 does not affect `away`. Line 4 doubles `away`'s own value, which is still 5.

Reread 1.4.2 *The Right Side Is Evaluated First* if your output differed.

Question 4

A. Line 2 evaluates `width * 3` once, when `width` is 4, and stores 12 in `area`. Changing `width` on line 3 does not rerun line 2.

- **B** is wrong: 30 is 10×3 , which assumes `area` recomputes itself when `width` changes, like a spreadsheet formula. It does not.
- **C** is wrong: 10 is the new value of `width`. Line 4 prints `area`, a separate variable.
- **D** is wrong: every statement is legal. Assigning a new value to `width` after using it is allowed.

Reread 1.4.2 *The Right Side Is Evaluated First* if you chose differently.

Question 5

B. Division comes before addition. `9 / 2` uses two `int` values, so it is the `int` 4. Then `4 + 0.5` involves a `double`, so the result is the `double` 4.5. Storing it with `double value = 9 / 2 + 0.5;` and printing `value` displays 4.5.

- **A** is wrong: 5.0 comes from computing `9 / 2` as 4.5. With two `int` values, the decimal part is dropped first.
- **C** is wrong: the last operation adds a `double`, so the whole expression is a `double`, not an `int`.
- **D** is wrong: it is neither the right type nor the right value. Nothing in the expression rounds 4.5 up.

Reread 1.4.3 *Every Expression Has a Type* if you chose differently.

Question 6

Corrected line 1: `double half = 9 / 2.0;`

The expression `9 / 2.0` includes a `double`, so its value is the `double` 4.5. The variable `half` is an `int`, and an `int` cannot hold the `.5`, so the compiler reports incompatible types: possible lossy conversion from `double` to `int`. Declaring `half` as a `double` makes the types compatible, and the code prints 4.5.

Changing the expression to `9 / 2` would compile, but it would store the `int` 4, not 4.5, so it does not do what the code is meant to do.

Reread 1.4.3 *Every Expression Has a Type* if you missed the error.

Question 7

B. `e` is declared but never assigned before `e + 2` uses it, so the compiler reports variable `e` might not have been initialized.

- **A** compiles: `a` is initialized on its second line, before `a + 1` uses it.
- **C** compiles: `c` is initialized to 0 in its declaration, so `c + 1` can use it. (It becomes 1.)
- **D** compiles: `d` is declared and then assigned. It is never used in an expression, and assigning to a variable is not "using" it.

Reread 1.4.4 *Assign Before You Use* if you chose differently.

Question 8

Model answer: If Java quietly used 0, the program would run and print an answer that looks reasonable but is wrong, because 0 was never the value the programmer intended. That is a logic error, which can go unnoticed. A compile error stops the program before it runs and points to the exact line, so the programmer must choose the correct starting value.

A complete answer must say that:

- a silent default could produce wrong results without any error message, and
- the compile error catches the mistake early and forces a deliberate choice.

Reread 1.4.4 *Assign Before You Use* if your answer differed.

Question 9

```
null
Hawks
null
```

Line	team holds	Printed
1	null	
2	null	null
3	connected to "Hawks"	
4	connected to "Hawks"	Hawks
5	null	
6	null	null

Line 5 shows that `null` can also replace an object. Assignment always replaces the old value, whatever kind it is.

Reread 1.4.5 *Reference Variables and null* if your output differed.

Question 10

C. `null` means "not connected to any object," so it can be stored only in a variable of a reference type. `String` is a reference type.

- **A** is wrong: `int` is primitive. Java reports incompatible types: `<null>` cannot be converted to `int`.
- **B** is wrong: `double` is primitive too, so it gets the same error, ending in `to double`.
- **D** is wrong: `boolean` is primitive and can hold only `true` or `false`. The error ends in `to boolean`.

Reread 1.4.5 *Reference Variables and null* if you chose differently.

Question 11

D. A `Scanner` reads text: characters such as the digits of a PIN.

- **A** is wrong: a camera image is visual input.
- **B** is wrong: a spoken passphrase is audio input.
- **C** is wrong: a thumb press is tactile input.

All four are input, because all four bring information into the program from outside. Only the typed PIN is text.

Reread 1.4.6 *Input and the Scanner Class* if you chose differently.

Question 12

Model answer: `new Scanner(System.in)` reads whatever the user types on the keyboard while the program runs, so the input, and therefore the output, can change every run. `new Scanner("16")` reads from the fixed text `16`, so the program reads the same input every time. The guide uses the string version so that every example has one exact output you can trace and check. The reading calls, such as `nextInt()`, work the same way with either one.

A complete answer must mention that:

- `System.in` is the keyboard and a string is fixed text, and
- fixed text makes the output predictable, while the reading works the same either way.

Reread 1.4.6 *Input and the Scanner Class* if your answer differed.

Question 13

```
14
2.5
```

Line	Call / work	Value stored	Input left
8	<code>input.nextInt()</code>	<code>first ← 8</code>	▸ 2.5 6
9	<code>input.nextDouble()</code>	<code>second ← 2.5</code>	▸ 6
10	<code>input.nextInt()</code>	<code>third ← 6</code>	(nothing)
11	<code>8 + 6 → 14</code>	<code>sum ← 14</code>	

The reads happen in order, so `third` gets the third piece, 6, not 2.5. `sum` adds `first` and `third` only.

Reread 1.4.7 *Reading Values with a Scanner* if your output differed.

Question 14

B. `nextLine()` reads everything up to the end of the current line, spaces included, and leaves out the line break. The first line is `Blue Team`. A following `input.nextInt()` would then read 7.

- **A** is wrong: `nextLine()` does not stop at a space. Stopping at the space is how the number-reading calls separate pieces.
- **C** is wrong: `nextLine()` stops at the line break, so the 7 on the next line is not included.
- **D** is wrong: reads happen in order from the start of the input, so the first call cannot skip to the 7.

Reread 1.4.7 *Reading Values with a Scanner* if you chose differently.

Question 15

B. `/` and `*` have the same precedence and come before `+`, and operators with the same precedence go left to right. `11 / 4` uses two `int` values, so it is the `int` 2. Then `2 * 2.0` involves a `double`, giving the `double` 4.0. Finally `4.0 + 1` is the `double` 5.0, which is stored in the `double` variable `result`.

- **A** is wrong: 6.5 treats `11 / 4` as 2.75. With two `int` values the decimal part is dropped before the multiplication.
- **C** is wrong: once `2.0` enters the calculation, every later result is a `double`, and a `double` prints with a decimal point.
- **D** is wrong: a `double` value stored in a `double` variable is compatible.

Reread 1.4.3 *Every Expression Has a Type* if you chose differently.

Question 16

```
10
9
```

Line	Work	first	second
1	Store 5	5	
2	Store 9	5	9
3	Copy second (9) into first	9	9

4	Copy first (now 9) into second	9	9
5	first + 1 → 9 + 1 → 10	10	9

The swap did **not** work. Line 3 replaced 5 with 9, and the 5 was gone for good, so line 4 had only 9 to copy. Each variable holds one value, and assignment replaces it. A swap needs a third variable to save one value first, such as `int saved = first;` before line 3, then `second = saved;` in place of line 4.

Reread *1.4.2 The Right Side Is Evaluated First* and *1.4.1 The Assignment Operator* if your output differed.

Question 17

D. `3.0 * 4` involves a `double`, so its value is the `double` 12.0. An `int` variable is not compatible with a `double` value, so Java reports possible lossy conversion from `double` to `int`, even though 12.0 has nothing after the decimal point. The compiler checks the type, not the particular value.

- **A** compiles: `String` is a reference type, so it can hold `null`.
- **B** compiles: `7 / 2.0` is the `double` 3.5, stored in a `double`.
- **C** compiles: `12 / 5` uses two `int` values, so it is the `int` 2, stored in an `int`.

Reread *1.4.3 Every Expression Has a Type* and *1.4.5 Reference Variables and null* if you chose differently.

Question 18

Model answer: In version 1, `motto` is declared but never assigned, so it has not been initialized. Java requires a value before a variable is used, and reports `variable motto might not have been initialized`. In version 2, `motto` is initialized with the value `null`. That is a real value meaning "no object," so the variable may be used, and printing it displays `null`.

A complete answer must say that:

- "never assigned" (no value) and "assigned `null`" (a value meaning no object) are different, and
- only an initialized variable can be used, which is why version 1 fails and version 2 runs.

Reread *1.4.4 Assign Before You Use* and *1.4.5 Reference Variables and null* if your answer differed.

Question 19

Error 1 (line 11): `int bonus = null;` tries to store `null` in an `int`. `null` means "no object" and fits only reference types, so Java reports incompatible types: `<null> cannot be converted to int`. "No bonus points" for a number is 0. Fix: `int bonus = 0;`

Error 2 (line 12): the right side, `(myScore + bonus) * 100.0 / maxPoints`, involves 100.0, so its value is a `double`. It cannot be stored in the `int` variable `percent`:
possible lossy conversion from `double` to `int`. Fix:

```
double percent = (myScore + bonus) * 100.0 / maxPoints;
```

With both fixes, line 12 evaluates $(17 + 0) * 100.0 / 20 \rightarrow 1700.0 / 20 \rightarrow 85.0$. Here is the corrected program (renamed so it can be saved alongside the original):

QuizScoresFixed.java

```

1  import java.util.Scanner;
2
3  public class QuizScoresFixed
4  {
5      public static void main(String[] args)
6      {
7          Scanner input = new Scanner("Unit 1 quiz\n17 20");
8          String quizName = input.nextLine();
9          int myScore = input.nextInt();
10         int maxPoints = input.nextInt();
11         int bonus = 0;
12         double percent = (myScore + bonus) * 100.0 / maxPoints;
13         System.out.println(quizName);
14         System.out.println(percent);
15     }
16 }

```

```

Unit 1 quiz
85.0

```

Reread 1.4.5 *Reference Variables and null* and 1.4.3 *Every Expression Has a Type* if you missed either error.

Question 20

```

Maple Street
3
2
35.0

```

Line	Call / work	Type	Stored
8	input.nextLine()	String	street ← "Maple Street"
9	input.nextInt()	int	boxes ← 14
10	input.nextInt()	int	helpers ← 4
11	input.nextDouble()	double	poundsPerBox ← 2.5
12	14 / 4 → 3 (integer division)	int	boxesEach ← 3
13	14 % 4 → 2 (remainder)	int	leftOver ← 2
14	14 * 2.5 → 35.0	double	totalPounds ← 35.0

Line 12 drops the decimal part of 3.5 because both values are `int`; the 2 boxes that do not divide evenly appear as the remainder on line 13. Line 14 has a `double`, so the result prints as 35.0.

Reread 1.4.7 *Reading Values with a Scanner* and 1.4.3 *Every Expression Has a Type* if your output differed.

Question 21

D. `nextInt()` can produce only a whole number. The next piece of input is 4.5, which is not one, so the `Scanner` throws an `InputMismatchException` when line 8 runs, and line 9 never runs.

- **A** is wrong: `nextInt()` does not drop the decimal part the way integer division does. It refuses input that is not a whole number.
- **B** is wrong: `hours` is an `int` and `nextInt()` produces an `int`, so 4.5 can never be stored or printed here.
- **C** is wrong: the compiler checks the code, not the input text. `int hours = input.nextInt();` stores an `int` in an `int`, which is compatible, so it compiles. The mismatch is discovered only while running.

Reread *1.4.7 Reading Values with a Scanner* if you chose differently.

Question 22

B. The source of the text changes, but the reading calls behave the same, which is why this guide's string-based examples show exactly how keyboard programs work.

- **A** is wrong: input can be tactile, audio, visual, or text, and `Scanner` is one way to get text input.
- **C** is wrong: without the `import`, the compiler reports `cannot find symbol` for `Scanner`.
- **D** is wrong: creating a `Scanner` only sets it up. Each value is read when a call such as `nextInt()` runs, one piece at a time, in order, and stored by that line's assignment.

Reread *1.4.6 Input and the Scanner Class* and *1.4.7 Reading Values with a Scanner* if you chose differently.