

3.8 Scope and Access

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

By the end you will be able to

- Explain what a variable's scope is and find the scope of any variable in a class.
- Explain why a local variable declared in one method, loop, or if statement cannot be used outside that block.
- Explain why constructor and method parameters are local variables, and store a parameter's value in an instance variable when other methods need it.
- Explain why local variables and parameters can never be marked public or private, while instance variables and methods can.
- Predict which variable a name refers to when a local variable or parameter has the same name as an instance variable, and fix the bugs this causes.
- Read the compiler messages cannot find symbol, already defined, has private access, and illegal start of expression, and fix the code behind them.

What to remember from earlier

- Private means only code in the declaring class may use something, and public means other classes may too. ([Topic 3.3: Anatomy of a Class, 3.3.3 The public and private Keywords](#))
- Each object has its own copy of every instance variable. ([Topic 3.3: Anatomy of a Class, 3.3.2 Each Object Has Its Own Instance Variables](#))
- An instance variable never assigned keeps its default value: 0, 0.0, false, or null. ([Topic 3.4: Constructors, 3.4.4 Default Values and the Default Constructor](#))
- Each parameter starts as a copy of its argument. ([Topic 3.5: Methods: How to Write Them, 3.5.6 Parameters Receive Copies](#))
- A non-void method returns a value with return. ([Topic 3.5: Methods: How to Write Them, 3.5.3 Non-Void Methods Return a Value](#))
- A variable declared in a for loop header or loop body cannot be used after the loop. ([Topic 2.8: for Loops](#))

How the examples in this chapter are laid out. Each example is one runnable file that holds two classes: a class with `main` that tries things out, and the class being studied. Java allows only one public class per file, so the second class is written as plain `class Trip` instead of `public class Trip`. In the course, and on the exam, every class is `public` and lives in its own file. Leaving `public` off here changes nothing this chapter teaches; in particular, `private` still keeps the other class out, as you will see in 3.8.4.

3.8.1 Scope and Local Variables

Picture a class with ten methods. Each method needs a few working variables: a total here, a counter there, a temporary result. If every one of those variables could be seen by every method, you would have to invent a different name for each, and a typo in one method could quietly

change a value another method depends on. Java avoids this by limiting where each variable can be used.

You have already met this limit. In Topic 2.8 you read that a variable declared in a for loop header "belongs to that loop" and cannot be used after it, and in Topic 2.10 that a counter declared inside a loop body "exists only inside those braces." This subsection gives that rule its real name and shows that it is not special to loops.

The part of a program where a variable's name can be used is called the variable's **scope**. Outside its scope, the name means nothing to Java, just as a locker number means nothing in a different school.

Scope is measured in blocks. A **block** is a section of code between a matching { and }. The body of a method is a block, the body of a constructor is a block, and so are the bodies of loops and if statements. A **local variable** is a variable declared inside a block (or in the header that opens a block, such as a for loop header). Its scope starts at its declaration and ends at the closing brace of the block it was declared in.

KEY IDEA

A local variable can be used only inside the block where it is declared, from its declaration to that block's closing brace.

WORKED EXAMPLE A local variable in each method

The `Trip` class stores a distance and a driving time. Look for the two variables that are declared *inside* methods: `speed` on line 24 and `gallons` on line 30.

TripReport.java

```
1 public class TripReport
2 {
3     public static void main(String[] args)
4     {
5         Trip drive = new Trip(150, 3);
6         drive.printSpeed();
7         drive.printFuel();
8     }
9 }
10
11 class Trip
12 {
13     private int miles;
14     private int hours;
15
16     public Trip(int distance, int time)
17     {
18         miles = distance;
19         hours = time;
20     }
21
22     public void printSpeed()
23     {
24         int speed = miles / hours;
```

```

25         System.out.println("Average speed: " + speed + " mph");
26     }
27
28     public void printFuel()
29     {
30         double gallons = miles / 30.0;
31         System.out.println("Fuel used: " + gallons + " gallons");
32     }
33 }

```

```

Average speed: 50 mph
Fuel used: 5.0 gallons

```

Here is what happens, in order:

1. Line 5 creates a `Trip` object. The constructor (lines 16–20) stores 150 in `miles` and 3 in `hours`.
2. Line 6 calls `printSpeed`. Line 24 creates a new local variable `speed` and stores `150 / 3`, which is 50. Line 25 prints it. At line 26, the method's closing brace, `speed` goes out of scope and the variable is gone.
3. Line 7 calls `printFuel`. Line 30 creates the local variable `gallons` and stores `150 / 30.0`, which is 5.0. Line 31 prints it, and at line 32 `gallons` is gone too.

Compare the two kinds of variables in this class:

Variable	Declared	Can be used
<code>miles</code>	line 13, in the class body	in every constructor and method
<code>hours</code>	line 14, in the class body	in every constructor and method
<code>speed</code>	line 24, inside <code>printSpeed</code>	lines 24–26 only
<code>gallons</code>	line 30, inside <code>printFuel</code>	lines 30–32 only

`main` is a method too, so `drive` (line 5) is a local variable of `main`. It can be used from line 5 to line 8 and nowhere in `Trip`.

What happens if a method reaches for another method's local variable? Change one thing: replace `printFuel` with a `printSummary` method that wants to reuse `speed`.

WORKED EXAMPLE Reaching into another method

`TripSummaryError.java`

```

1  public class TripSummaryError
2  {
3      public static void main(String[] args)
4      {
5          Trip drive = new Trip(150, 3);
6          drive.printSpeed();
7          drive.printSummary();
8      }
9  }
10
11  class Trip

```

```
12 {
13     private int miles;
14     private int hours;
15
16     public Trip(int distance, int time)
17     {
18         miles = distance;
19         hours = time;
20     }
21
22     public void printSpeed()
23     {
24         int speed = miles / hours;
25         System.out.println("Average speed: " + speed + " mph");
26     }
27
28     public void printSummary()
29     {
30         System.out.println(miles + " miles at " + speed + " mph");
31     }
32 }
```

This does not compile. Java reports:

```
TripSummaryError.java:30: error: cannot find symbol
    System.out.println(miles + " miles at " + speed + " mph");
                                   ^
symbol:   variable speed
location: class Trip
```

(The exact wording can vary between Java versions.) "Symbol" is the compiler's word for a name. On line 30 Java looks for something called `speed` that is in scope there. `miles` is fine, because instance variables can be used in every method. But the only `speed` in the file was declared on line 24, inside `printSpeed`, and its scope ended at line 26. Inside `printSummary` there is no such name.

Notice that this is a *compile-time* error. Java checks scope by reading the code, before anything runs. It does not matter that `printSpeed` is called before `printSummary` on lines 6 and 7; `speed` is still out of scope on line 30.

The fix depends on what you want. To show the speed in both methods, compute it again in `printSummary` with its own local variable, or make the value an instance variable so every method can reach it.

Because each method's local variables belong only to that method's block, two methods may use the same local variable name without any conflict. They are two different variables that happen to share a name.

WORKED EXAMPLE Two variables named total

Both methods of `Cart` declare a local variable called `total`. Watch whether changing one affects the other.

TwoTotals.java

```
1 public class TwoTotals
2 {
3     public static void main(String[] args)
4     {
5         Cart cart = new Cart(4, 2.5);
6         System.out.println(cart.itemCost());
7         System.out.println(cart.costWithShipping());
8         System.out.println(cart.itemCost());
9     }
10 }
11
12 class Cart
13 {
14     private int items;
15     private double price;
16
17     public Cart(int count, double each)
18     {
19         items = count;
20         price = each;
21     }
22
23     public double itemCost()
24     {
25         double total = items * price;
26         return total;
27     }
28
29     public double costWithShipping()
30     {
31         double total = items * price;
32         total = total + 5.0;
33         return total;
34     }
35 }
```

```
10.0
15.0
10.0
```

- Line 6 calls `itemCost`. Line 25 creates a `total` holding `4 * 2.5`, which is `10.0`, and returns it.
- Line 7 calls `costWithShipping`. Line 31 creates a *different* `total`, also `10.0`, and line 32 adds `5.0` to it, making `15.0`.
- Line 8 calls `itemCost` again. A brand-new `total` is created on line 25 and holds `10.0`. Adding `5` to the other method's `total` had no effect here.

The last bullet shows something about time as well as place. A local variable is created when its declaration runs and disappears when its block ends. When the method is called again, the declaration runs again and makes a fresh variable. A local variable cannot remember anything from one call to the next. Instance variables can, because they belong to the object and last as long as the object does.

WATCH OUT

Expecting a local variable to remember its value between calls. This `Page` class is supposed to count how many times it has been visited:

`VisitCounterBug.java`

```
1 public class VisitCounterBug
2 {
3     public static void main(String[] args)
4     {
5         Page home = new Page();
6         System.out.println(home.visit());
7         System.out.println(home.visit());
8         System.out.println(home.visit());
9     }
10 }
11
12 class Page
13 {
14     public int visit()
15     {
16         int visits = 0;
17         visits++;
18         return visits;
19     }
20 }
```

```
1
1
1
```

Students write this because the counting pattern from Unit 2 looks right: start at 0, then add 1. But `visits` is declared on line 16, inside `visit`. Every call runs line 16 again, creating a new `visits` that starts at 0, so every call returns 1. The fix is to make the count an instance variable, which belongs to the object and survives between calls:

VisitCounterFixed.java

```
1 public class VisitCounterFixed
2 {
3     public static void main(String[] args)
4     {
5         Page home = new Page();
6         System.out.println(home.visit());
7         System.out.println(home.visit());
8         System.out.println(home.visit());
9     }
10 }
11
12 class Page
13 {
14     private int visits;
15
16     public Page()
17     {
18         visits = 0;
19     }
20
21     public int visit()
22     {
23         visits++;
24         return visits;
25     }
26 }
```

```
1
2
3
```

Now the constructor sets `visits` to 0 once, and each call to `visit` adds 1 to the same variable. If a value must outlive a single method call, it belongs in an instance variable, not a local variable.

PAUSE AND TRY

Question 1 Multiple choice

What happens when this file is compiled and run?

TimerQuestion.java

```
1 public class TimerQuestion
2 {
3     public static void main(String[] args)
4     {
5         Timer oven = new Timer(150);
6         oven.printMinutes();
7         oven.printLeftover();
8     }
```

```
9  }
10
11  class Timer
12  {
13      private int seconds;
14
15      public Timer(int s)
16      {
17          seconds = s;
18      }
19
20      public void printMinutes()
21      {
22          int minutes = seconds / 60;
23          System.out.println(minutes);
24      }
25
26      public void printLeftover()
27      {
28          int leftover = seconds - minutes * 60;
29          System.out.println(leftover);
30      }
31  }
```

- A. It prints 2 and then 30.
- B. It does not compile, because `minutes` on line 28 is out of scope.
- C. It prints 2 and then 150, because `minutes` is 0 inside `printLeftover`.
- D. It does not compile, because `seconds` is used in two different methods.

Question 2 Tracing

What does this program print? Keep track of `presses` and `total` on each call.

ClickerTrace.java

```
1 public class ClickerTrace
2 {
3     public static void main(String[] args)
4     {
5         Clicker counter = new Clicker();
6         System.out.println(counter.click());
7         System.out.println(counter.click());
8         System.out.println(counter.click());
9     }
10 }
11
12 class Clicker
13 {
14     private int total;
15
16     public Clicker()
17     {
18         total = 0;
19     }
20
21     public int click()
22     {
23         int presses = 0;
24         presses++;
25         total++;
26         return presses + total;
27     }
28 }
```

3.8.2 Blocks Inside Methods

A method body is the biggest block inside a method, but it is rarely the only one. As soon as a method contains a loop or an if statement, it has blocks inside blocks, and the scope rule from 3.8.1 applies to every one of them.

Think of nested blocks as rooms inside a house with one-way windows. Code in an inner room can see everything declared in the rooms around it, so a loop body can use the method's variables. Code in an outer room cannot see into an inner one, so the method cannot use a variable declared inside the loop body. (Unlike real windows, the view stops at the method's walls: one method still cannot see another method's variables.)

A variable declared in a for loop header is a special case worth naming: its scope is the header plus the loop body, and it ends when the loop does.

WORKED EXAMPLE Mapping every variable's scope

The method `countLetter` counts how many times a one-letter string appears in the word. It has variables at four different levels.

CountLetter.java

```

1  public class CountLetter
2  {
3      public static void main(String[] args)
4      {
5          Word fruit = new Word("banana");
6          System.out.println(fruit.countLetter("a"));
7      }
8  }
9
10 class Word
11 {
12     private String text;
13
14     public Word(String t)
15     {
16         text = t;
17     }
18
19     public int countLetter(String letter)
20     {
21         int count = 0;
22         for (int i = 0; i < text.length(); i++)
23         {
24             String current = text.substring(i, i + 1);
25             if (current.equals(letter))
26             {
27                 count++;
28             }
29         }
30         return count;
31     }
32 }

```

The output is 3, since "banana" has three a's. Now find each variable's block:

Variable	Declared on	Its block	Usable on lines
text	12	the whole class	11–32
count	21	the method body (20–31)	21–31
i	22	the for loop (22–29)	22–29
current	24	the loop body (23–29)	24–29

(The parameter `letter` is in scope for the whole method; 3.8.3 explains why.)

Line 27, `count++`, sits three blocks deep: inside the `if` body, inside the loop body, inside the method body. It can still use `count`, because `count` was declared in an outer block that is still open. Line 30 is back in the method body only. It can use `count`, but `i` and `current` are out of scope there.

Notice also that `current` is created fresh on every pass through the loop. Line 24 runs six times, and each time it makes a new `current` for that pass, just as each method call makes new local variables.

Now suppose you want the position of the *last* match instead of a count. The tempting code records the position at the moment it finds a match:

WORKED EXAMPLE A variable trapped inside an if body

FindLastError.java

```
1 public class FindLastError
2 {
3     public static void main(String[] args)
4     {
5         Word fruit = new Word("banana");
6         System.out.println(fruit.findLast("a"));
7     }
8 }
9
10 class Word
11 {
12     private String text;
13
14     public Word(String t)
15     {
16         text = t;
17     }
18
19     public int findLast(String letter)
20     {
21         for (int i = 0; i < text.length(); i++)
22         {
23             String current = text.substring(i, i + 1);
24             if (current.equals(letter))
25             {
26                 int position = i;
27             }
28         }
29         return position;
30     }
31 }
```

This does not compile:

```
FindLastError.java:29: error: cannot find symbol
    return position;
           ^
symbol:   variable position
location: class Word
```

`position` is declared on line 27, inside the if body (lines 26–28). Its scope ends at line 28. Line 29 is outside that block, so Java cannot find the name. There is also a logic problem hiding here: if the letter never appears, line 27 never runs, so there would be nothing to return.

The fix is to declare `position` in the block where it is needed at the end, the method body, before the loop starts. Inside the if body, assign to it instead of declaring it again:

FindLastFixed.java

```
1 public class FindLastFixed
2 {
3     public static void main(String[] args)
4     {
5         Word fruit = new Word("banana");
6         System.out.println(fruit.findLast("a"));
7         System.out.println(fruit.findLast("z"));
8     }
9 }
10
11 class Word
12 {
13     private String text;
14
15     public Word(String t)
16     {
17         text = t;
18     }
19
20     public int findLast(String letter)
21     {
22         int position = -1;
23         for (int i = 0; i < text.length(); i++)
24         {
25             String current = text.substring(i, i + 1);
26             if (current.equals(letter))
27             {
28                 position = i;
29             }
30         }
31         return position;
32     }
33 }
```

```
5
-1
```

For "a", line 28 runs at positions 1, 3, and 5, and each assignment replaces the last, so 5 is returned. For "z", line 28 never runs, so `position` keeps its starting value -1, a common signal for "not found." Declaring a variable before a block is how you carry a result out of it.

Since a variable disappears when its block ends, its name is free again afterward. Two loops one after the other may each declare their own `i`:

```
1 for (int i = 1; i <= 3; i++)
2 {
3     System.out.println(i);
4 }
5 for (int i = 10; i <= 30; i += 10)
6 {
7     System.out.println(i);
8 }
```

```
1  
2  
3  
10  
20  
30
```

The first `i` goes out of scope at the end of the first loop (line 4), so line 5 may declare a new `i`. The two never exist at the same time.

The rule is different for blocks *inside* one another. While a variable is in scope, an inner block may not declare another variable with the same name. Java refuses because a name in the inner block would be ambiguous: which one did you mean?

WATCH OUT

Reusing a name that is still taken. Favorite names like `i` get reused without a second thought. Here `i` was already declared for another purpose before the loop:

```
1 int i = 0;  
2 for (int i = 1; i <= 3; i++)  
3 {  
4     System.out.println(i);  
5 }
```

```
error: variable i is already defined in method main(String[])
```

`i` from line 1 is still in scope on line 2, because the method body has not ended. Line 2 tries to create a second `i` in a place where the first one is visible, so Java stops with `already defined`. (Wording can vary between Java versions.) The fix is to choose a different name for one of them, or, if you meant to use the existing variable, drop the type so the statement assigns instead of declaring.

PAUSE AND TRY

Question 3 Debugging

This code is supposed to keep a running total and print 10, 30, and 60. It does not compile. Find the error and correct the line.

```
1 int total = 0;  
2 for (int day = 1; day <= 3; day++)  
3 {  
4     int total = total + day * 10;  
5     System.out.println(total);  
6 }
```

Question 4 Multiple choice

In the corrected `findLast` method from `FindLastFixed.java` (above), which variables could be used on line 31, `return position;`?

- A. position and i only
- B. position, letter, and text
- C. position, i, current, letter, and text
- D. text only

3.8.3 Parameters Are Local Variables

When a constructor runs, its parameters hold the values the caller supplied. Can a method called later still use them? Answering that question correctly is the difference between a class that works and one full of `cannot find symbol` errors.

Recall from Topic 3.5 that a parameter is declared in a method's (or constructor's) header, and that each call starts the parameter off as a copy of the argument. The header is the line that opens the body's block, just as a `for` loop header opens the loop's block. So Java treats parameters exactly like local variables declared at the very top of the body: a **parameter is a local variable** whose scope is its constructor's or method's body, and nothing else.

KEY IDEA

A parameter is a local variable of its constructor or method. It can be used anywhere in that body and nowhere outside it, and a new copy is made on every call.

WORKED EXAMPLE Constructor and method parameters

Look at the two parameters, `startTemp` (line 16) and `degrees` (line 21), and where each one is used.

ThermostatDemo.java

```

1  public class ThermostatDemo
2  {
3      public static void main(String[] args)
4      {
5          Thermostat home = new Thermostat(68);
6          home.raise(4);
7          home.raise(2);
8          System.out.println("Target: " + home.getTarget());
9      }
10 }
11
12 class Thermostat
13 {
14     private int target;
15
16     public Thermostat(int startTemp)
17     {
18         target = startTemp;
19     }
20
21     public void raise(int degrees)
22     {
23         target = target + degrees;
24     }
25
26     public int getTarget()
27     {
28         return target;
29     }
30 }

```

Target: 74

Step	What happens	target
line 5	constructor runs; startTemp is 68; line 18 copies it to target	68
line 6	raise runs with degrees equal to 4	72
line 7	raise runs again with a new degrees, equal to 2	74
line 8	getTarget returns target	74

startTemp exists only while the constructor runs. Line 18 is what keeps its value alive: it copies the value into the instance variable `target`, which every method can use. Each call to `raise` gets its own `degrees`, and that `degrees` is gone when the call ends.

Change one thing: add a `reset` method that should put the thermostat back to the temperature it started at.

WORKED EXAMPLE Using a constructor parameter in a method

ThermostatResetError.java

```

1  public class ThermostatResetError

```

```
2 {
3     public static void main(String[] args)
4     {
5         Thermostat home = new Thermostat(68);
6         home.raise(4);
7         home.reset();
8         System.out.println("Target: " + home.getTarget());
9     }
10 }
11
12 class Thermostat
13 {
14     private int target;
15
16     public Thermostat(int startTemp)
17     {
18         target = startTemp;
19     }
20
21     public void raise(int degrees)
22     {
23         target = target + degrees;
24     }
25
26     public void reset()
27     {
28         target = startTemp;
29     }
30
31     public int getTarget()
32     {
33         return target;
34     }
35 }
```

ThermostatResetError.java:28: error: cannot find symbol

```
    target = startTemp;
```

^

symbol: variable startTemp

location: class Thermostat

`startTemp` is a local variable of the constructor (lines 16–19). Line 28 is inside `reset`, a different block, so the name is out of scope. Students make this mistake because the constructor and `reset` sit in the same class, and it feels as if the class should remember what it was given. It only remembers what was stored in an instance variable.

So store it. Add an instance variable for the starting temperature, and fill it in the constructor:

ThermostatResetFixed.java

```
1 public class ThermostatResetFixed
2 {
3     public static void main(String[] args)
4     {
```

```
5     Thermostat home = new Thermostat(68);
6     home.raise(4);
7     System.out.println("Target: " + home.getTarget());
8     home.reset();
9     System.out.println("Target: " + home.getTarget());
10 }
11 }
12
13 class Thermostat
14 {
15     private int target;
16     private int startingTarget;
17
18     public Thermostat(int startTemp)
19     {
20         target = startTemp;
21         startingTarget = startTemp;
22     }
23
24     public void raise(int degrees)
25     {
26         target = target + degrees;
27     }
28
29     public void reset()
30     {
31         target = startingTarget;
32     }
33
34     public int getTarget()
35     {
36         return target;
37     }
38 }
```

```
Target: 72
Target: 68
```

Line 21 copies the parameter into `startingTarget`, so `reset` (line 31) can use it long after the constructor has finished.

Because a parameter and a local variable declared at the top of the body share the same block, the "already defined" rule from 3.8.2 applies between them too.

WATCH OUT

Declaring a local variable with the same name as a parameter.

ParamRedeclare.java

```
1 public class ParamRedeclare
2 {
3     public static void main(String[] args)
4     {
5         Thermostat home = new Thermostat(68);
6         home.raise(4);
7         System.out.println("Target: " + home.getTarget());
8     }
9 }
10
11 class Thermostat
12 {
13     private int target;
14
15     public Thermostat(int startTemp)
16     {
17         target = startTemp;
18     }
19
20     public void raise(int degrees)
21     {
22         int degrees = 5;
23         target = target + degrees;
24     }
25
26     public int getTarget()
27     {
28         return target;
29     }
30 }
```

ParamRedeclare.java:22: error: variable degrees is already defined in method raise(int)

The parameter `degrees` (line 20) is already a variable in `raise`'s body, so line 22 cannot declare another one. Often the student meant to try a test value; in that case, pass the value as the argument (`home.raise(5);`) instead of overwriting the parameter. Otherwise, give the local variable a different name.

PAUSE AND TRY**Question 5** Debugging

This program should print `Welcome, Ada! Level 3`, but it does not compile. Find the error, explain it, and correct the line.

PlayerNameError.java

```
1 public class PlayerNameError
2 {
3     public static void main(String[] args)
4     {
5         Player hero = new Player("Ada", 3);
6         hero.greet();
7     }
8 }
9
10 class Player
11 {
12     private String name;
13     private int level;
14
15     public Player(String playerName, int startLevel)
16     {
17         name = playerName;
18         level = startLevel;
19     }
20
21     public void greet()
22     {
23         System.out.println("Welcome, " + playerName + "! Level " + level);
24     }
25 }
```

Question 6 Short answer

A programmer adds this method to the `Thermostat` class from `ThermostatDemo.java`, next to `raise`:

```
1 public void lower(int degrees)
2 {
3     target = target - degrees;
4 }
```

A classmate predicts a compile error, because `raise` already has a parameter named `degrees`. Are they right? Explain using scope, and say what `Target: line` prints if `main` creates new `Thermostat(68)`, calls `raise(4)`, then calls `lower(6)`.

3.8.4 Instance Variables and Access from Other Classes

So far, local variables have had small scopes, and instance variables have been usable "in every method." It is time to say exactly why, and to connect scope with the access keywords from Topic 3.3.

An instance variable is declared in the class body, outside every constructor and method. The class body is the outermost block of the class, and every constructor and method sits inside it. So by the same inner-sees-outer rule from 3.8.2, every constructor and method can use every

instance variable. The same holds for the class's methods: any method of the class can call any other method of the class.

Scope answers "where in *this class* can the name be used?" A second question remains: can code in a *different* class use it, through an object? That is what the access keywords control. Recall from Topic 3.3 that the keywords `public` and `private` (called **access modifiers**) are written in front of instance variables, constructors, and methods. `private` limits use to code written inside the declaring class. `public` also allows use from other classes.

KEY IDEA

Instance variables and methods can be used anywhere inside their class. Whether another class can use them, through an object and the dot operator, depends on `public` or `private`.

WORKED EXAMPLE Using an account only through its public methods

The `Account` class has a `private` instance variable, two `public` methods, and one `private` helper method.

BankDemo.java

```
1 public class BankDemo
2 {
3     public static void main(String[] args)
4     {
5         Account checking = new Account(50.0);
6         checking.deposit(25.5);
7         checking.deposit(-10.0);
8         System.out.println(checking.getBalance());
9     }
10 }
11
12 class Account
13 {
14     private double balance;
15
16     public Account(double opening)
17     {
18         balance = opening;
19     }
20
21     public void deposit(double amount)
22     {
23         if (isPositive(amount))
24         {
25             balance = balance + amount;
26         }
27     }
28
29     public double getBalance()
30     {
31         return balance;
32     }
33 }
```

```
34     private boolean isPositive(double amount)
35     {
36         return amount > 0;
37     }
38 }
```

75.5

1. Line 5 creates an account with balance equal to 50.0.
2. Line 6 calls `deposit(25.5)`. Inside `Account`, line 23 calls the private method `isPositive`. That is allowed: the call is written inside the class. It returns `true`, so line 25 makes balance 75.5.
3. Line 7 calls `deposit(-10.0)`. `isPositive` returns `false`, so balance is unchanged.
4. Line 8 calls the public method `getBalance`, which returns 75.5.

`main` never touches `balance` directly; it goes through `deposit` and `getBalance`. This is encapsulation from Topic 3.3: the class decides how its data may change, and here it refuses negative deposits.

What if `main` skips the methods and reads the variable itself?

WORKED EXAMPLE Reaching for private members from another class

BankPrivateError.java

```
1  public class BankPrivateError
2  {
3      public static void main(String[] args)
4      {
5          Account checking = new Account(50.0);
6          checking.deposit(25.5);
7          System.out.println(checking.balance);
8      }
9  }
10
11 class Account
12 {
13     private double balance;
14
15     public Account(double opening)
16     {
17         balance = opening;
18     }
19
20     public void deposit(double amount)
21     {
22         if (isPositive(amount))
23         {
24             balance = balance + amount;
25         }
26     }
27
28     public double getBalance()
```

```

29     {
30         return balance;
31     }
32
33     private boolean isPositive(double amount)
34     {
35         return amount > 0;
36     }
37 }

```

```

BankPrivateError.java:7: error: balance has private access in Account
    System.out.println(checking.balance);
                        ^

```

Line 7 is inside `BankPrivateError`, a different class from `Account`. `balance` is private, so only code inside `Account` may use it. The message is different from `cannot find symbol: Java did find balance`, but this class is not allowed to use it. Fix it by calling the public accessor, `checking.getBalance()`.

Private methods are protected the same way. Calling `isPositive` from `main` fails with a matching message:

`BankPrivateMethodError.java`

```

1  public class BankPrivateMethodError
2  {
3      public static void main(String[] args)
4      {
5          Account checking = new Account(50.0);
6          if (checking.isPositive(20.0))
7          {
8              checking.deposit(20.0);
9          }
10         System.out.println(checking.getBalance());
11     }
12 }
13
14 class Account
15 {
16     private double balance;
17
18     public Account(double opening)
19     {
20         balance = opening;
21     }
22
23     public void deposit(double amount)
24     {
25         if (isPositive(amount))
26         {
27             balance = balance + amount;
28         }
29     }

```

```

30
31     public double getBalance()
32     {
33         return balance;
34     }
35
36     private boolean isPositive(double amount)
37     {
38         return amount > 0;
39     }
40 }

```

BankPrivateMethodError.java:6: error: isPositive(double) has private access in Account

(Wording can vary between Java versions.) This second class was written without `public` so the example fits in one file, and `private` still blocked access, which shows the rule is about the declaring class, not about files.

Two different checks can stop a name, so keep them apart:

Question	Decided by	Error when it fails
Is the name in scope here?	where it was declared	cannot find symbol
May this class use that object's member?	public or private	... has private access

WATCH OUT

Using an instance variable's name alone in another class. Inside `Account`, `balance` by itself means "this account's balance." In `main`, writing `balance` alone (with no object and dot) is not a request for any account's balance; it is just an unknown name, so Java reports `cannot find symbol` rather than `has private access`. Another class can only reach a member through an object, as in `checking.getBalance()`, and only if that member is `public`.

PAUSE AND TRY

Question 7 Multiple choice

Here is the `Ticket` class and a `main` method with one blank. Which expression, written in the blank, compiles?

```
1 public class TicketQuestion
2 {
3     public static void main(String[] args)
4     {
5         Ticket show = new Ticket(12.0);
6         System.out.println( /* expression goes here */ );
7     }
8 }
9
10 class Ticket
11 {
12     private double price;
13
14     public Ticket(double cost)
15     {
16         price = cost;
17     }
18
19     public double getPrice()
20     {
21         return price;
22     }
23
24     private double discount()
25     {
26         return price * 0.5;
27     }
28 }
```

- A. `show.price`
- B. `show.getPrice()`
- C. `show.discount()`
- D. `price`

Question 8 Short answer

In `BankDemo.java`, line 25 inside `deposit` uses `balance`, but line 7 of `BankPrivateError.java` cannot use `checking.balance`. Explain the difference.

3.8.5 No Access Keywords on Local Variables

If `private` protects instance variables so well, why not mark a method's local variables `private` too, just to be safe? Java will not let you, and the reason follows directly from scope.

An access modifier answers one question: which *classes* may use this member through an object? That question only makes sense for things that belong to the class or object and last beyond one method call: instance variables, constructors, and methods (and the class variables from Topic 3.7). A local variable is already far more restricted than `private`. It cannot be used outside its own block, not even by another method of the same class, and it disappears when the block ends. No

other class could ever reach it, so there is nothing for `public` or `private` to decide. Because the keyword would be meaningless, Java treats it as a mistake.

KEY IDEA

Local variables, including parameters, can never be declared `public` or `private`. Their scope already limits them to one block.

WORKED EXAMPLE An access keyword on a local variable

The programmer wanted the bonus to stay hidden, so they added `private`:

PrivateLocal.java

```
1 public class PrivateLocal
2 {
3     public static void main(String[] args)
4     {
5         Quiz test = new Quiz(18);
6         System.out.println(test.finalScore());
7     }
8 }
9
10 class Quiz
11 {
12     private int points;
13
14     public Quiz(int earned)
15     {
16         points = earned;
17     }
18
19     public int finalScore()
20     {
21         private int bonus = 2;
22         return points + bonus;
23     }
24 }
```

```
PrivateLocal.java:21: error: illegal start of expression
    private int bonus = 2;
    ^
```

Inside a method body, Java expects statements, and a statement cannot begin with `private`. The compiler gets confused at that word, reports `illegal start of expression`, and often prints more errors for the following lines (here one about line 22) that disappear once the first is fixed. Always fix the first error first. The fix is to delete `private`: write `int bonus = 2;`. The program then prints 20. `bonus` is still hidden from every other class, and from every other method of `Quiz`, by its scope alone.

Parameters are local variables too, so the same rule applies to them. Here only the parameter changed:

WORKED EXAMPLE An access keyword on a parameter

PrivateParam.java

```

1  public class PrivateParam
2  {
3      public static void main(String[] args)
4      {
5          Quiz test = new Quiz(18);
6          test.addPoints(3);
7          System.out.println(test.getPoints());
8      }
9  }
10
11 class Quiz
12 {
13     private int points;
14
15     public Quiz(int earned)
16     {
17         points = earned;
18     }
19
20     public void addPoints(private int extra)
21     {
22         points = points + extra;
23     }
24
25     public int getPoints()
26     {
27         return points;
28     }
29 }

```

```

PrivateParam.java:20: error: modifier private not allowed here
    public void addPoints(private int extra)
                           ^

```

The method itself may be `public`, because other classes call it. The parameter `extra` may not be marked, because it is local to `addPoints`. Remove `private` from inside the parentheses. (The message differs from the previous example because Java notices the problem in a different place; wording can also vary between Java versions.)

What it is	Declared where	Scope	public or private allowed?
instance variable	class body, outside every method	the whole class	yes (normally <code>private</code>)
method or constructor	class body	the whole class	yes
parameter	constructor or method header	that constructor or method body	no
local variable	inside a block	declaration to end of the block	no

WATCH OUT

Copying an instance variable declaration into a method. Students often write the instance variables first, `private int bonus;`, then later decide `bonus` is only needed in one method and cut and paste the whole line into the method, `private` included. When you move a variable into a method, remove the access modifier.

PAUSE AND TRY**Question 9** Multiple choice

Each line below is the first statement inside a method body, followed by `System.out.println(rate);`. Which one compiles?

- A. `private double rate = 0.5;`
- B. `double rate = 0.5;`
- C. `public double rate = 0.5;`
- D. `private rate = 0.5;`

3.8.6 When a Local Name Matches an Instance Variable

Naming a constructor parameter after the instance variable it fills in seems like the clearest choice: the parameter for `coins` should be called `coins`, right? It compiles, too. But the program can then quietly do the wrong thing.

From 3.8.2, Java does not allow a local variable to reuse the name of *another local variable* that is still in scope. Instance variables are different: they are not local, so a local variable or parameter *may* have the same name as an instance variable. When that happens, there are two variables with one name in scope at the same time, and Java needs a rule to decide which one the name means.

The rule: inside that constructor or method, wherever the local variable or parameter is in scope, the name refers to the **local variable** (or parameter), not the instance variable. The closer declaration wins. Programmers say the local variable **shadows** the instance variable: it stands in front of it and hides it, the way a person standing close to you can block your view of someone farther away.

KEY IDEA

When a local variable or parameter has the same name as an instance variable, that name means the local variable or parameter everywhere it is in scope. The instance variable is hidden there.

WORKED EXAMPLE A constructor parameter that shadows

CoinJarBug.java

```
1 public class CoinJarBug
2 {
3     public static void main(String[] args)
4     {
5         CoinJar jar = new CoinJar(40);
6         System.out.println("Coins: " + jar.getCoins());
7     }
8 }
9
10 class CoinJar
11 {
12     private int coins;
13
14     public CoinJar(int coins)
15     {
16         coins = coins;
17     }
18
19     public int getCoins()
20     {
21         return coins;
22     }
23 }
```

Coins: 0

Follow the name `coins` through the program:

Line	Code	Which coins?	Effect
12	<code>private int coins;</code>	the instance variable	starts at the default value, 0
14	<code>CoinJar(int coins)</code>	the parameter	holds 40 during the constructor
16	<code>coins = coins;</code>	both sides are the parameter	copies 40 into itself; nothing changes
21	<code>return coins;</code>	the instance variable (no parameter here)	returns 0

Line 16 looked like "store the parameter in the instance variable," but because the parameter shadows the instance variable inside the constructor, both names refer to the parameter. The instance variable is never assigned, so it keeps its default value 0 from Topic 3.4. The constructor ends, the parameter disappears, and the 40 is lost. Java gives no error, because assigning a variable to itself is legal.

The simplest fix is a parameter name that does not clash:

```
1 public CoinJar(int startCoins)
2 {
3     coins = startCoins;
4 }
```

With that change, the program prints `Coins: 40`. Topic 3.9 introduces the keyword `this`, which gives another way to reach a shadowed instance variable.

Shadowing does not need a parameter. Change one thing: here the name clash comes from a local variable declared in a constructor.

WORKED EXAMPLE A local variable that shadows

ScoreboardBug.java

```
1 public class ScoreboardBug
2 {
3     public static void main(String[] args)
4     {
5         Scoreboard game = new Scoreboard();
6         System.out.println("Lives: " + game.getLives());
7         game.loseLife();
8         System.out.println("Lives: " + game.getLives());
9     }
10 }
11
12 class Scoreboard
13 {
14     private int lives;
15     private int score;
16
17     public Scoreboard()
18     {
19         int lives = 3;
20         score = 0;
21     }
22
23     public void loseLife()
24     {
25         lives--;
26     }
27
28     public int getLives()
29     {
30         return lives;
31     }
32 }
```

```
Lives: 0
Lives: -1
```

The programmer meant line 19 to set the starting lives to 3. But `int lives = 3;` has a type in front, so it *declares a new local variable* named `lives`, which shadows the instance variable for the rest of the constructor. The 3 goes into the local variable, which disappears at line 21. The instance variable `lives` is untouched and keeps its default 0. Then `loseLife` (line 25), where no local `lives` exists, subtracts 1 from the instance variable, giving -1.

Compare line 20, `score = 0;`, which has no type and so assigns to the instance variable as intended. The fix for line 19 is to remove `int`:

```
1      lives = 3;
```

With that change, the program prints `Lives: 3` and then `Lives: 2`.

WATCH OUT

Shadowing compiles, so the compiler cannot warn you. Every other mistake in this chapter stops the program at compile time. Shadowing does not; the program runs and prints a wrong value, often a default like `0`, `0.0`, or `null`. When an instance variable seems to "not get set," check two things in the constructor or method that should set it:

1. Is there a parameter with the same name as the instance variable?
2. Does the assignment start with a type (`int lives = 3;`), which declares a new local variable?

Either one makes the name refer to a local variable instead of the instance variable.

PAUSE AND TRY

Question 10 Tracing

What does this program print?

TankTrace.java

```
1 public class TankTrace
2 {
3     public static void main(String[] args)
4     {
5         Tank truck = new Tank(50);
6         System.out.println(truck.afterTrip(30));
7         System.out.println(truck.getFuel());
8     }
9 }
10
11 class Tank
12 {
13     private int fuel;
14
15     public Tank(int startFuel)
16     {
17         fuel = startFuel;
18     }
19
20     public int afterTrip(int fuel)
21     {
22         fuel = fuel - 10;
23         return fuel;
24     }
25
26     public int getFuel()
27     {
28         return fuel;
29     }
30 }
```

Question 11 Debugging

This program should print Age: 5, but it prints Age: 2. Find the error, explain it, and correct the code.

PetAgeBug.java

```
1 public class PetAgeBug
2 {
3     public static void main(String[] args)
4     {
5         Pet dog = new Pet(2);
6         dog.setAge(5);
7         System.out.println("Age: " + dog.getAge());
8     }
9 }
10
11 class Pet
12 {
13     private int age;
14
15     public Pet(int startAge)
16     {
17         age = startAge;
18     }
19
20     public void setAge(int age)
21     {
22         age = age;
23     }
24
25     public int getAge()
26     {
27         return age;
28     }
29 }
```

Putting it together

This program tracks a summer reading goal. It uses every kind of variable from this chapter. Before reading the trace, try to list each variable's scope yourself.

ReadingLog.java

```
1 public class ReadingLog
2 {
3     public static void main(String[] args)
4     {
5         BookLog summer = new BookLog(20);
6         summer.logDay(25);
7         summer.logDay(12);
8         summer.logDay(20);
9         System.out.println(summer.summary());
10    }
11 }
12
13 class BookLog
```

```

14 {
15     private int goal;
16     private int days;
17     private int totalPages;
18     private int goalDays;
19
20     public BookLog(int pagesPerDay)
21     {
22         goal = pagesPerDay;
23         days = 0;
24         totalPages = 0;
25         goalDays = 0;
26     }
27
28     public void logDay(int pages)
29     {
30         days++;
31         totalPages += pages;
32         if (pages >= goal)
33         {
34             goalDays++;
35         }
36     }
37
38     public String summary()
39     {
40         String stars = "";
41         for (int i = 0; i < goalDays; i++)
42         {
43             stars += "*";
44         }
45         double average = (double) totalPages / days;
46         return days + " days, " + average + " pages per day " + stars;
47     }
48 }

```

3 days, 19.0 pages per day **

Step 1: map the scopes.

Variable	Kind	Usable on lines
summer	local variable of main	5–10
goal, days, totalPages, goalDays	private instance variables	all of BookLog (14–48); never in main
pagesPerDay	constructor parameter	21–26
pages	parameter of logDay	29–36
stars	local variable of summary	40–47
i	for loop variable	41–44
average	local variable of summary	45–47

No local name matches an instance variable, so nothing is shadowed: every use of `goal`, `days`, `totalPages`, and `goalDays` inside `BookLog` reaches the instance variables. Line 22 is what lets `logDay` compare against the goal later, since `pagesPerDay` disappears when the constructor ends.

Step 2: trace the calls.

Call (line)	Parameter	days	totalPages	goalDays
constructor (5)	<code>pagesPerDay = 20</code>	0	0	0
<code>logDay</code> (6)	<code>pages = 25</code>	1	25	1
<code>logDay</code> (7)	<code>pages = 12</code>	2	37	1
<code>logDay</code> (8)	<code>pages = 20</code>	3	57	2

Each call to `logDay` gets a fresh `pages`, but the instance variables carry their values from call to call. On the third call, `20 >= 20` is true, so the goal counts.

Step 3: the summary. Line 9 calls `summary`. Line 40 creates `stars` as `""`. The loop runs `goalDays` times (twice), so `stars` becomes `"**"`; `i` goes out of scope at line 44. Line 45 creates `average` as `57.0 / 3`, which is `19.0`. Line 46 builds and returns the text, and line 9 prints it. When `summary` returns, `stars` and `average` are gone, while the instance variables remain in the object for any later call.

What one small change would break. If line 22 were written `int goal = pagesPerDay;`, the constructor would declare a local `goal` that shadows the instance variable (3.8.6). The instance `goal` would stay 0, every day would meet the goal, and the output would end with `***`. And if `main` tried to print `summer.days`, Java would report `days` has private access in `BookLog` (3.8.4).

Common mistakes

Most mistakes in this chapter show up as compiler messages. Use this table to go from the message to the cause.

What you see	Usual cause	Fix
cannot find symbol for a variable	used outside its block: another method's local, a parameter of another method, or a loop variable after the loop	declare it in the block that needs it, before any inner block; or store it in an instance variable
variable ... is already defined	declared a second variable with a name that is still in scope (an outer local or a parameter)	rename it, or drop the type to assign to the existing variable
... has private access in ...	another class used a private instance variable or method through an object	call a public method instead
illegal start of expression modifier ... not allowed or here	public or private on a local variable or parameter	delete the access modifier
No error, but an instance variable stays 0, 0.0, or null	a parameter or local variable with the same name shadows it	rename the parameter, or remove the type from the assignment

Vocabulary

Term	Definition	Example
scope	The part of a program where a variable's name can be used.	<code>speed</code> can be used only on lines 24–26 of <code>TripReport.java</code>
block	A section of code between a matching <code>{</code> and <code>}</code> , such as a method, constructor, loop, or if body.	the body of a for loop
local variable	A variable declared inside a block or a block's header; usable only from its declaration to the end of that block.	<code>int count = 0;</code> inside a method
parameter (as a local variable)	A variable declared in a constructor or method header; local to that constructor or method body.	<code>degrees</code> in <code>raise(int degrees)</code>
access modifier	The keyword <code>public</code> or <code>private</code> on an instance variable, constructor, or method; never on a local variable.	<code>private double balance;</code>
shadow	When a local variable or parameter has the same name as an instance variable, the name refers to the local one and hides the instance variable.	<code>coins = coins;</code> in a constructor with parameter <code>coins</code>

Chapter summary

- A variable's **scope** is where its name can be used. A local variable's scope runs from its declaration to the closing brace of the block it was declared in, and Java checks this at compile time (`cannot find symbol`).
- Blocks nest: inner blocks can use variables from the blocks around them, but not the other way around. A name can be reused in separate blocks, but not declared again while it is still in scope (`already defined`).
- Parameters are local variables of their constructor or method. To keep a parameter's value for other methods, copy it into an instance variable. Local variables are created fresh on each call, so they cannot remember values between calls.
- Instance variables and methods can be used anywhere in their class; `private` stops other classes from using them (`has private access`). Local variables and parameters can never be `public` or `private`.
- When a local variable or parameter has the same name as an instance variable, the name means the local one there. This compiles, so the symptom is an instance variable that never changes from its default.

Review questions

Question 12 Multiple choice

In `Playlist`, which variables can be used inside `lengthText` (lines 28–32)?

`PlaylistChoice.java`

```
1 public class PlaylistChoice
2 {
3     public static void main(String[] args)
4     {
5         Playlist mix = new Playlist();
6         mix.addSong(200);
7         System.out.println(mix.lengthText());
8     }
9 }
10
11 class Playlist
12 {
13     private int songs;
14     private int totalSeconds;
15
16     public Playlist()
17     {
18         songs = 0;
19         totalSeconds = 0;
20     }
21
22     public void addSong(int seconds)
23     {
24         songs++;
25         totalSeconds += seconds;
26     }
27
28     public String lengthText()
29     {
30         int minutes = totalSeconds / 60;
31         return songs + " songs, " + minutes + " min, last song " + seconds + " s";
32     }
33 }
```

- A. `songs`, `totalSeconds`, `minutes`, and `seconds`
- B. `songs`, `totalSeconds`, and `minutes`
- C. `minutes` only
- D. `seconds` and `minutes` only

Question 13 Tracing

What does this program print?

`GardenTrace.java`

```
1 public class GardenTrace
2 {
3     public static void main(String[] args)
4     {
```

```
5      Garden yard = new Garden(10);
6      yard.plantRows(3);
7      System.out.println("Total: " + yard.getPlants());
8  }
9  }
10
11  class Garden
12  {
13      private int plants;
14
15      public Garden(int start)
16      {
17          plants = start;
18      }
19
20      public void plantRows(int rows)
21      {
22          int plants = 0;
23          for (int r = 1; r <= rows; r++)
24          {
25              plants += 4;
26          }
27          System.out.println("Planted: " + plants);
28      }
29
30      public int getPlants()
31      {
32          return plants;
33      }
34  }
```

Question 14 Debugging

This program should print the runner's fastest time, Best: 11.8. It prints Best: 0.0. Find the error, explain why it produces 0.0 exactly, and correct the line.

RaceBestBug.java

```
1  public class RaceBestBug
2  {
3      public static void main(String[] args)
4      {
5          Runner sam = new Runner();
6          sam.record(12.5);
7          sam.record(11.8);
8          sam.record(12.1);
9          System.out.println("Best: " + sam.getBest());
10     }
11 }
12
13 class Runner
14 {
15     private double bestTime;
16
17     public Runner()
```

```
18     {
19         double bestTime = 99.0;
20     }
21
22     public void record(double time)
23     {
24         if (time < bestTime)
25         {
26             bestTime = time;
27         }
28     }
29
30     public double getBest()
31     {
32         return bestTime;
33     }
34 }
```

Question 15 Short answer

A classmate writes `private int i = 0;` as the first line of a method, then uses `i` as a loop counter. They explain, "I made it private so that no other class can change my counter." What happens, and what is wrong with their reasoning?

Question 16 Multiple choice

Each segment below is the whole body of a `main` method. Which segments compile?

Segment I

```
1  int score = 85;
2  if (score >= 80)
3  {
4      String message = "Great job";
5      System.out.println(message);
6  }
7  if (score < 90)
8  {
9      String message = "Keep going";
10     System.out.println(message);
11 }
```

Segment II

```
1  int total = 0;
2  for (int i = 1; i <= 3; i++)
3  {
4      int doubled = i * 2;
5      total += doubled;
6  }
7  System.out.println(total);
```

Segment III

```
1 String word = "hop";
2 for (int k = 0; k < word.length(); k++)
3 {
4     String word = word.substring(k, k + 1);
5     System.out.println(word);
6 }
```

- A. I only
- B. I and II only
- C. II and III only
- D. I, II, and III

Question 17 Tracing

What does this program print? Notice that `main` and `read` both have a variable named `pages`.

SameNameTrace.java

```
1 public class SameNameTrace
2 {
3     public static void main(String[] args)
4     {
5         int pages = 10;
6         Book novel = new Book(300);
7         novel.read(pages);
8         System.out.println(pages);
9         System.out.println(novel.getPagesLeft());
10    }
11 }
12
13 class Book
14 {
15     private int pagesLeft;
16
17     public Book(int total)
18     {
19         pagesLeft = total;
20     }
21
22     public void read(int pages)
23     {
24         pagesLeft = pagesLeft - pages;
25         pages = 0;
26     }
27
28     public int getPagesLeft()
29     {
30         return pagesLeft;
31     }
32 }
```

Reflection

- ☐ I can explain what scope is and find the lines where a local variable can be used. Reread: 3.8.1
Scope and Local Variables
- ☐ I can explain why a local variable cannot remember its value from one method call to the next.
Reread: 3.8.1 Scope and Local Variables
- ☐ I can tell which variables are in scope inside nested loop and if blocks, and when a name may be reused. Reread: 3.8.2 Blocks Inside Methods
- ☐ I can explain why a parameter cannot be used in another method, and how to keep its value.
Reread: 3.8.3 Parameters Are Local Variables
- ☐ I can tell the difference between cannot find symbol and has private access. Reread: 3.8.4
Instance Variables and Access from Other Classes
- ☐ I can explain why local variables and parameters are never public or private. Reread: 3.8.5 No
Access Keywords on Local Variables
- ☐ I can predict which variable a shadowed name refers to and fix the bug it causes. Reread: 3.8.6
When a Local Name Matches an Instance Variable

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. `minutes` is a local variable declared on line 22, inside `printMinutes`. Its scope ends at line 24. Line 28 is in a different method, so Java reports `cannot find symbol` for `minutes` and the program never runs.

- **A** is what the programmer intended (`150 / 60` is 2, and `150 - 120` is 30), but it assumes `printLeftover` can see `printMinutes`'s variable. It cannot.
- **C** is wrong: Java does not invent a `minutes` holding 0. An out-of-scope name is a compile error, not a variable with a default value.
- **D** is wrong: `seconds` is an instance variable, declared in the class body on line 13, so every method of `Timer` may use it.

Reread 3.8.1 *Scope and Local Variables* if you chose differently.

Question 2

2
3
4

`total` is an instance variable, so it keeps its value between calls. `presses` is a local variable, so line 23 makes a new one starting at 0 on every call.

Call	<code>presses</code> after line 24	<code>total</code> after line 25	Returned
1st	1	1	2
2nd	1	2	3
3rd	1	3	4

If you answered 2, 4, 6, you treated `presses` as if it remembered its value. Reread 3.8.1 *Scope and Local Variables* if your output differed.

Question 3

Line 4 declares a second `total`. The `total` from line 1 is declared in the outer block and is still in scope inside the loop body, so `int total = ...` on line 4 tries to create another variable with the same name where the first is visible. Java reports `variable total is already defined`. Change line 4 so it assigns to the existing variable:

```
1    total = total + day * 10;
```

Now each pass adds to the one `total`: 0 + 10 is 10, 10 + 20 is 30, and 30 + 30 is 60. (Even if Java had allowed a second `total` inside the loop, it would be a new variable on every pass, so the running total would never build up.)

Reread 3.8.2 *Blocks Inside Methods* if you found a different error.

Question 4

B. Line 31 is in the method body, outside the loop. Variables in scope there are those declared in the method body before line 31 (`position`), the method's parameter (`letter`), and the instance variable (`text`), which every method can use.

- **A** is wrong: `i` was declared in the loop header, so its scope ended with the loop on line 30. And it leaves out `letter` and `text`.
- **C** is wrong: `i` and `current` belong to the loop and the loop body, which have both closed by line 31.
- **D** is wrong: local variables declared earlier in the same method body, such as `position`, are still in scope on line 31.

Reread 3.8.2 *Blocks Inside Methods* if you chose differently.

Question 5

Line 23 uses `playerName`, which is out of scope. `playerName` is a parameter of the constructor, so it is a local variable whose scope is the constructor body (lines 16–19). Java reports cannot find symbol for `playerName` on line 23. The constructor already copied the value into the instance variable `name` on line 17, and instance variables can be used in every method, so use `name`:

```
1      System.out.println("Welcome, " + name + "! Level " + level);
```

This prints `Welcome, Ada! Level 3`.

Reread 3.8.3 *Parameters Are Local Variables* if you found a different error.

Question 6

Model answer: No, it compiles. Each parameter is a local variable of its own method, so `raise's degrees` has the scope of `raise's` body and `lower's degrees` has the scope of `lower's` body. The two bodies are separate blocks (neither is inside the other), so the two variables are never in scope at the same place; they are different variables that share a name. The program prints Target: 66: 68 + 4 is 72, then 72 - 6 is 66. Each call gets its own fresh `degrees`, while `target` is the instance variable shared by both methods.

A complete answer says: parameters are local to their method; the two methods are separate blocks; so the same name is allowed.

Reread 3.8.3 *Parameters Are Local Variables* and the two-totals example in 3.8.1 if your answer differed.

Question 7

B. `getPrice` is `public`, so code in another class may call it through an object. It prints `12.0`.

- **A** is wrong: `price` is `private`, so Java reports `price` has private access in `Ticket`.
- **C** is wrong: `discount` is a `private` method. It can be called inside `Ticket`, but not from `main`. Java reports `discount()` has private access in `Ticket`.
- **D** is wrong: in `main`, `price` alone is not a name that is in scope at all, so Java reports cannot find symbol. Instance variables are in scope only inside their own class.

Reread 3.8.4 *Instance Variables and Access from Other Classes* if you chose differently.

Question 8

Model answer: `balance` is declared in the body of `Account`, so it is in scope in every method of `Account`, including `deposit`. Because `deposit` is written inside `Account`, `private` does not stop it. Line 7 of `BankPrivateError.java` is in a different class. Java can find `balance` through the object checking, but `private` means only code inside `Account` may use it, so Java reports `has private access`.

A complete answer says: the instance variable's scope is its whole class; `private` allows use only inside the declaring class; `main` is in another class.

Reread *3.8.4 Instance Variables and Access from Other Classes* if your answer differed.

Question 9

B. A local variable is declared with just its type and name. This compiles and prints `0.5`.

- **A** is wrong: local variables cannot be `private`. Java reports `illegal start of expression`.
- **C** is wrong for the same reason. `public` would try to let other classes use a variable that only exists inside one method, which makes no sense. Java reports `illegal start of expression`.
- **D** is wrong twice over: it uses an access modifier on a local variable, and it leaves out the type. Java reports `illegal start of expression`.

Reread *3.8.5 No Access Keywords on Local Variables* if you chose differently.

Question 10

```
20
50
```

The constructor (line 17) sets the instance variable `fuel` to 50. In `afterTrip`, the parameter is also named `fuel`, so on lines 22 and 23 `fuel` means the parameter, which starts at 30. Line 22 makes the parameter 20, and line 23 returns 20. The instance variable is never touched. `getFuel` has no parameter named `fuel`, so its `fuel` is the instance variable, still 50.

If you answered 20 and 20, you assumed line 22 changed the instance variable. Reread *3.8.6 When a Local Name Matches an Instance Variable* if your output differed.

Question 11

The parameter of `setAge` (line 20) shadows the instance variable. Inside `setAge`, `age` means the parameter, so line 22, `age = age;`, copies the parameter into itself. The instance variable keeps the 2 it got from the constructor, and line 7 prints `Age: 2`. Java gives no error because the code is legal. Rename the parameter and assign it to the instance variable:

```
1      public void setAge(int newAge)
2      {
3          age = newAge;
4      }
```

Now `age` on line 22 is the instance variable, it becomes 5, and the program prints `Age: 5`.

Reread *3.8.6 When a Local Name Matches an Instance Variable* if you found a different error.

Question 12

B. `songs` and `totalSeconds` are instance variables, in scope everywhere in the class. `minutes` is declared on line 30, inside `lengthText`. `seconds` is a parameter of `addSong`, so it is local to `addSong` (lines 23–26). As written, line 31 uses `seconds` and the file does not compile: Java reports `cannot find symbol for seconds`.

- **A** is wrong: it includes `seconds`, assuming a value passed to one method can be seen by another.
- **C** is wrong: it forgets that instance variables can be used in every method of their class.
- **D** is wrong: it includes the out-of-scope parameter and leaves out the instance variables.

To show the last song's length, the class would need an instance variable such as `lastSong`, set in `addSong`. Reread 3.8.3 *Parameters Are Local Variables* and 3.8.4 if you chose differently.

Question 13

```
Planted: 12
Total: 10
```

The constructor sets the instance variable `plants` to 10. In `plantRows`, line 22 declares a *local* variable `plants` (it has a type), which shadows the instance variable for the rest of the method. The loop runs for `r` equal to 1, 2, and 3, adding 4 to the local `plants` each time: 4, 8, 12. Line 27 prints the local value, `Planted: 12`. At line 28 the local `plants` disappears. `getPlants` has no local `plants`, so it returns the instance variable, still 10.

If you answered `Total: 22`, you assumed line 25 added to the instance variable. Reread 3.8.6 *When a Local Name Matches an Instance Variable* and 3.8.2 *Blocks Inside Methods* if your output differed.

Question 14

Line 19 declares a local variable instead of setting the instance variable.

`double bestTime = 99.0;` starts with a type, so it creates a new local `bestTime` that shadows the instance variable inside the constructor. The 99.0 goes into the local variable, which disappears at line 20. The instance variable `bestTime` is never assigned, so it keeps the default value for a `double`, 0.0. In `record`, no time is less than 0.0, so `bestTime` never changes, and the program prints `Best: 0.0`. Remove the type:

```
1      bestTime = 99.0;
```

Now `bestTime` starts at 99.0. `12.5 < 99.0` makes it 12.5, `11.8 < 12.5` makes it 11.8, and `12.1 < 11.8` is false, so the program prints `Best: 11.8`.

Reread 3.8.6 *When a Local Name Matches an Instance Variable* if you found a different error.

Question 15

Model answer: The code does not compile; Java reports an error such as `illegal start of expression` because `private` cannot be used on a local variable. Their goal is already met without it: `i` is a local variable, so its scope is limited to that method's body. No other class, and not even another method of the same class, can use it. Access modifiers are only for instance variables, constructors, methods, and class variables, which other classes could otherwise reach through an object or the class name. The fix is to write `int i = 0;`.

A complete answer says: it is a compile error; local variables cannot be `public` or `private`; scope already keeps the variable inside its block.

Reread 3.8.5 *No Access Keywords on Local Variables* if your answer differed.

Question 16

B. Segment I compiles: the two `message` variables are in separate if blocks that are not nested, so each one's scope ends before the other begins. It prints `Great job` and `Keep going`. Segment II compiles: the loop body is inside the method body, so it may use and change `total`; `doubled` is used only inside the loop body. It prints `12`. Segment III does not compile: `word` from line 1 is still in scope inside the loop body, so line 4 cannot declare another `word`, and Java reports `variable word is already defined`. (Renaming the inner variable, for example to `letter`, fixes it.)

- **A** is wrong: it rejects Segment II, perhaps thinking a loop body cannot change a variable declared outside it. Inner blocks can use outer variables.
- **C** is wrong: it rejects Segment I and accepts Segment III, reversing the rule. Reusing a name is fine in separate blocks, not while the first variable is still in scope.
- **D** is wrong: Segment III declares `word` again while the first `word` is still in scope.

Reread 3.8.2 *Blocks Inside Methods* if you chose differently.

Question 17

```
10
290
```

Step	pages in main	pages in read	pagesLeft
lines 5–6	10	(does not exist)	300
call on line 7	10	10 (a copy)	300
line 24	10	10	290
line 25	10	0	290
back in main	10	(gone)	290

The two `pages` variables are local to different methods, so they are separate variables. The call copies `10` into `read`'s parameter (Topic 3.5). Line 25 changes only that copy, which disappears when `read` ends. `main`'s `pages` is still `10`. The instance variable `pagesLeft` was changed on line 24, and the object keeps that change, so `getPagesLeft` returns `290`.

Reread 3.8.3 *Parameters Are Local Variables* if your output differed.