

3.3 Anatomy of a Class

Self-Study Guide · About 50 minutes · With answer key

By the end you will be able to

- Name the parts of a class (header, instance variables, constructors, methods) and find each one in real code.
- Explain why every object built from a class has its own copy of each instance variable, and trace code that changes them.
- Decide whether a line of code can reach a variable, constructor, or method, based on the words `public` and `private`.
- Explain what encapsulation is and why instance variables are usually `private`.
- Explain why classes and constructors are `public` in this course, and when a method should be `public` or `private`.

What to remember from earlier

- An attribute whose value can differ for each object is an instance variable. ([Topic 3.1: Abstraction and Program Design](#), [3.1.4 Instance Variables and Class Variables](#))
- Writing `new` with a constructor call builds an object and gives back a reference to it. ([Topic 1.13: Object Creation and Storage \(Instantiation\)](#), [1.13.1 Constructors and the `new` Keyword](#))
- A variable of a class type holds a reference. ([Topic 1.12: Objects: Instances of Classes](#), [1.12.4 Reference Variables](#))
- Assigning one reference variable to another copies the reference, so both name the same object. ([Topic 1.12: Objects: Instances of Classes](#), [1.12.5 Assigning and Copying References](#))
- You call a method on an object with a dot. ([Topic 1.14: Calling Instance Methods](#), [1.14.2 The Dot Operator](#))
- A method header gives the return type, the method name, and the parameters. ([Topic 1.9: Method Signatures](#), [1.9.5 Method Signatures](#))

3.3.1 The Parts of a Class

In Topic 3.1 you designed classes on paper: a list of attributes and a list of behaviors. Until now, every class you used, such as `String` and `Scanner`, was written by someone else, and you only saw its documentation. To turn your own design into something Java can build objects from, you need to know what a class looks like on the inside: which pieces it has, where each one goes, and how Java tells them apart.

Think of a class file as a form with labeled sections. The top line says what the form is for. Below it come the sections, in a usual order: first the data every object keeps, then the code that builds a new object, then the actions an object can perform.

WORKED EXAMPLE A scoreboard class, part by part

Here is a complete program. The class being studied is `Scoreboard`, which starts on line 10. Lines 1 to 8 are a small `GameNight` class whose `main` method builds one `Scoreboard` object and asks it to print. Read the `Scoreboard` part first.

GameNight.java

```
1 public class GameNight
2 {
3     public static void main(String[] args)
4     {
5         Scoreboard gym = new Scoreboard();
6         gym.printScores();
7     }
8 }
9
10 class Scoreboard
11 {
12     private String homeTeam;
13     private int homeScore;
14     private int awayScore;
15
16     public Scoreboard()
17     {
18         homeTeam = "Hawks";
19         homeScore = 0;
20         awayScore = 0;
21     }
22
23     public void printScores()
24     {
25         System.out.println(homeTeam + " " + homeScore + ", Visitors " +
26         awayScore);
27     }
28 }
```

Running it prints:

```
Hawks 0, Visitors 0
```

Now take `Scoreboard` apart, from the top down.

Lines	Part
10	The class header : the keyword <code>class</code> and the class name
11–27	The class body, between one pair of braces; everything that belongs to the class goes inside them
12–14	Three instance variables , one for each attribute; each is declared inside the class but outside any constructor or method
16–21	The constructor ; it has the same name as the class and runs when <code>new Scoreboard()</code> builds an object
23–26	A method , <code>printScores</code> ; it is a behavior any <code>Scoreboard</code> object can perform

Follow the program as Java runs it. Line 5 calls the constructor with `new Scoreboard()`. Java builds a `Scoreboard` object that has its own `homeTeam`, `homeScore`, and `awayScore`, then runs lines 18 to 20, which give them their starting values "Hawks", 0, and 0. The reference to the new object is stored in `gym`. Line 6 calls `printScores` on that object, so line 25 runs. Inside the class, code can use an instance variable by its name alone, so `homeTeam` on line 25 means "this object's `homeTeam`". The text joined by `+` (Topic 1.15) is `Hawks 0, Visitors 0`.

The words `private` and `public` in front of the variables, the constructor, and the method control which code is allowed to use each piece. They are the main subject of this chapter, starting in 3.3.3; for now, read past them. You read constructors in Topic 1.13, and Topic 3.4 teaches you to write them fully; here every constructor simply gives each instance variable a starting value. In the same way, Topic 3.5 teaches you to write methods fully. The methods in this chapter are short `void` methods with no parameters, so you can focus on how a class is organized.

A **class** is declared with the keyword `class`, followed by the class name, followed by a body in braces. In this course, every class is also marked `public`, so on the AP exam the header of this class would read `public class Scoreboard`. The usual order inside the body is instance variables first, then constructors, then methods. Java does not require that order, but nearly all Java code follows it, so a reader always knows where to look.

Why this program's Scoreboard is not marked public

Look again at line 10: it says `class Scoreboard`, not `public class Scoreboard`. That is because of a Java rule about files: **one file can hold at most one public class, and that class must have the same name as the file**. `GameNight.java` already holds the `public class GameNight`, so `Scoreboard` cannot be `public` in the same file.

Every runnable example in this chapter puts two classes in one file this way so that you can see the whole program at once. Real projects, and AP exam questions, put each class in its own file, named after the class, and mark every class `public`:

```
1 // Scoreboard.java
2 public class Scoreboard
3 {
4     // instance variables, constructors, and methods go here
5 }
```

```
1 // GameNight.java
2 public class GameNight
3 {
4     public static void main(String[] args)
5     {
6         Scoreboard gym = new Scoreboard();
7         gym.printScores();
8     }
9 }
```

When those two files sit in the same project folder, `GameNight` can use `Scoreboard` exactly as before.

KEY IDEA

A class has a header (`public class Name`) and a body in braces that holds, in order, its instance variables, its constructors, and its methods. In this course every class is public, and each class normally lives in its own file with the same name.

WORKED EXAMPLE A second class with different parts

Classes differ in how many of each part they have, but the layout stays the same. Here is a `Thermostat` class written the way it would be in its own file, `Thermostat.java`:

```
1 public class Thermostat
2 {
3     private double currentTemp;
4     private double targetTemp;
5     private boolean heaterOn;
6
7     public Thermostat()
8     {
9         currentTemp = 18.5;
10        targetTemp = 21.0;
11        heaterOn = false;
12    }
13
14    public void printReading()
15    {
16        System.out.println(currentTemp);
17    }
18
19    public void raiseTarget()
20    {
21        targetTemp = targetTemp + 0.5;
22    }
23 }
```

- The header is `public class Thermostat`.
- There are three instance variables, of three different types: two `double`s and a `boolean`. Each one is a single declaration with no starting value, ending in a semicolon.
- The constructor is the block named `Thermostat`, the same name as the class. It sets all three starting values.
- There are two methods this time, `printReading` and `raiseTarget`. The second one changes an instance variable instead of printing it: every call adds `0.5` to that object's `targetTemp`.

How do you tell the constructor from a method? The constructor's name is the class name, and its header has no `void` (or any other return type) in it. The methods have their own names and the word `void` before the name.

WATCH OUT

Marking both classes public in one file. Since every class in this course is public, it is tempting to write `public` on both classes of a one-file program:

TwoPublicClasses.java

```
1 public class TwoPublicClasses
2 {
3     public static void main(String[] args)
4     {
5         Scoreboard gym = new Scoreboard();
6         gym.printScores();
7     }
8 }
9
10 public class Scoreboard
11 {
12     private int homeScore;
13
14     public Scoreboard()
15     {
16         homeScore = 0;
17     }
18
19     public void printScores()
20     {
21         System.out.println(homeScore);
22     }
23 }
```

This does not compile. Java reports (wording can vary between Java versions):

```
TwoPublicClasses.java:10: error: class Scoreboard is public, should be declared in a
file named Scoreboard.java
```

Java uses the file name to find a public class, so a public class named Scoreboard has to live in Scoreboard.java. There are two fixes. In a one-file program, leave public off the second class, as this chapter's examples do. In a real project, move Scoreboard into its own file, Scoreboard.java, where it can be public.

PAUSE AND TRY**Question 1** Multiple choice

Look at this class:

```
1 public class Lamp
2 {
3     private boolean isOn;
4     private int brightness;
5
6     public Lamp()
7     {
8         isOn = false;
9         brightness = 50;
10    }
11
12    public void switchOn()
13    {
14        isOn = true;
15    }
16 }
```

Which statement correctly describes its parts?

- A. It has one instance variable, `isOn`, and two methods, `Lamp` and `switchOn`.
- B. It has two instance variables, one constructor, and one method.
- C. It has two instance variables and two constructors, `Lamp` and `switchOn`.
- D. It has four instance variables, because `isOn` and `brightness` each appear twice.

Question 2 Short answer

A student puts a `public class Lamp` and a `public class LampTest` (which has `main`) together in one file named `LampTest.java`, and it does not compile. Explain why, and describe the fix a real project would use.

3.3.2 Each Object Has Its Own Instance Variables

A basketball league tracks many players, and every player has a name and a point total. The class `Player` describes what every player has. But Ana's points are not Ben's points. If Ana scores, Ben's total must not change. Where does each player's total live, so that the totals stay separate?

The answer is that instance variables belong to the object, not to the class. The class declares that every player *has* a `points` variable. Each time `new Player()` builds an object, that object gets its own fresh copy of `points`, and of every other instance variable. A paper form works the same way: the blank form says "Name: __ **Points:** __", and every copy of the form you hand out has its own blanks to fill in.

KEY IDEA

Instance variables belong to the object. Every object of a class has its own copy of each instance variable, so changing one object's copy has no effect on any other object.

To show this, the `Player` class in the next examples marks its instance variables `public`, which lets code outside the class reach them directly. Outside code names an object's instance variable the same way it calls an object's method in Topic 1.14: the reference, a dot, then the name. `ana.points`

means "the `points` variable that belongs to the object `ana` refers to." (3.3.4 explains why real classes usually do not make instance variables `public`.)

WORKED EXAMPLE Two players, two point totals

Watch which object each line changes.

TwoPlayers.java

```

1 public class TwoPlayers
2 {
3     public static void main(String[] args)
4     {
5         Player ana = new Player();
6         Player ben = new Player();
7         ana.name = "Ana";
8         ana.points = 12;
9         ben.name = "Ben";
10        ben.points = 7;
11        ana.points = ana.points + 3;
12        System.out.println(ana.name + " has " + ana.points);
13        System.out.println(ben.name + " has " + ben.points);
14    }
15 }
16
17 class Player
18 {
19     public String name;
20     public int points;
21
22     public Player()
23     {
24         name = "Nobody";
25         points = 0;
26     }
27 }
```

Lines 5 and 6 call the constructor twice, so Java builds two separate `Player` objects. Each runs lines 24 and 25 for itself, so both start with `name` set to "Nobody" and `points` set to 0. Memory now looks like this:

ana → Player object	ben → Player object
name: "Nobody"	name: "Nobody"
points: 0	points: 0

Now trace `main`:

Line	Statement	ana's object	ben's object
7	<code>ana.name = "Ana";</code>	"Ana", 0	"Nobody", 0
8	<code>ana.points = 12;</code>	"Ana", 12	"Nobody", 0
9	<code>ben.name = "Ben";</code>	"Ana", 12	"Ben", 0
10	<code>ben.points = 7;</code>	"Ana", 12	"Ben", 7
11	<code>ana.points = ana.points + 3;</code>	"Ana", 15	"Ben", 7

Line 11 reads `ana`'s `points` (12), adds 3, and stores 15 back into `ana`'s `points`. `ben`'s object is a different object with its own `points`, so it still holds 7. The program prints:

```
Ana has 15
Ben has 7
```

Notice the count: there is only one `Player` class, written once, yet there are two `points` variables, because there are two objects. The number of copies of an instance variable equals the number of objects built, not the number of classes.

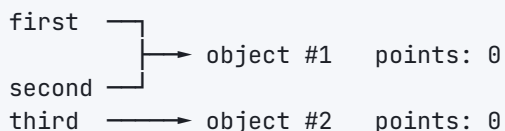
WORKED EXAMPLE Two variables, one object

What if two variables name the *same* object? Recall from Topic 1.13 that assigning one reference variable to another copies the reference, not the object. This program builds only two objects but has three variables.

SamePlayer.java

```
1 public class SamePlayer
2 {
3     public static void main(String[] args)
4     {
5         Player first = new Player();
6         Player second = first;
7         Player third = new Player();
8         second.points = 20;
9         third.points = 4;
10        System.out.println(first.points);
11        System.out.println(second.points);
12        System.out.println(third.points);
13    }
14 }
15
16 class Player
17 {
18     public String name;
19     public int points;
20
21     public Player()
22     {
23         name = "Nobody";
24         points = 0;
25     }
26 }
```

- Line 5 builds object #1; `first` refers to it.
- Line 6 copies the reference, so `second` refers to object #1 too. No new object is built, because there is no `new`.
- Line 7 builds object #2; `third` refers to it.



Line 8 sets `second.points` to 20. That is object #1's points. Line 9 sets object #2's points to 4. So `first.points` and `second.points` are the same variable, and both print 20:

```

20
20
4

```

The rule did not change: each object has its own copy. What changed is the number of objects. `first` and `second` share a copy because they share an object; `third` has its own because it refers to its own object.

WATCH OUT

Using the class name instead of an object. Because `points` is declared in the `Player` class, students sometimes write `Player.points`:

ClassNameField.java

```

1 public class ClassNameField
2 {
3     public static void main(String[] args)
4     {
5         Player ana = new Player();
6         Player.points = 12;
7         System.out.println(ana.points);
8     }
9 }
10
11 class Player
12 {
13     public String name;
14     public int points;
15
16     public Player()
17     {
18         name = "Nobody";
19         points = 0;
20     }
21 }

```

This does not compile. Java reports (wording can vary):

```

ClassNameField.java:6: error: non-static variable points cannot be referenced from a
static context

```

The message uses the word `static`, which Topic 3.7 explains; what matters here is the idea behind it. There is no single `points` that belongs to the class. Every `Player` object has its own,

so `Player.points` does not say *which* player's points to change. Java needs a reference to one object. The fix is to name the object: `ana.points = 12`; compiles and the program prints 12.

PAUSE AND TRY

Question 3 Tracing

Using the `Player` class from this subsection, what does this program print?

`PlayerTrace.java`

```
1 public class PlayerTrace
2 {
3     public static void main(String[] args)
4     {
5         Player p = new Player();
6         Player q = new Player();
7         Player r = q;
8         p.points = 5;
9         q.points = 9;
10        r.points = r.points + p.points;
11        p.points = 1;
12        System.out.println(p.points);
13        System.out.println(q.points);
14        System.out.println(r.points);
15    }
16 }
17
18 class Player
19 {
20     public String name;
21     public int points;
22
23     public Player()
24     {
25         name = "Nobody";
26         points = 0;
27     }
28 }
```

Question 4 Multiple choice

A program runs these statements with the `Player` class from this subsection:

```
1 Player a = new Player();
2 Player b = new Player();
3 Player c = new Player();
4 Player d = b;
```

How many `points` variables exist in memory?

- A. 1
- B. 3

- C. 4
- D. 5

3.3.3 The public and private Keywords

In 3.3.2, `main` could change any player's points to anything it liked. That is convenient for a demonstration, but think about a real score-keeping app with code written by several programmers. Any line, in any class, could set a player's points to `-500`. Java gives the author of a class a way to say which pieces outside code may use and which pieces are for the class's own code only.

That is the job of the words in front of each declaration. They are **access modifiers**: keywords that decide which code is allowed to use a class, a variable, a constructor, or a method. This course uses two:

- **public**: any code may use it, inside or outside the class that declares it.
- **private**: only code inside the class that declares it may use it. Code in any other class cannot, not even by naming the object with a dot.

Here, "inside the class" means between that class's braces: its constructors and its methods. Code in any other class, including the `main` method of a different class, is outside.

Where the code is	Can use a public member?	Can use a private member?
A constructor or method of the same class	Yes	Yes
Code in a different class (such as another class's <code>main</code>)	Yes	No; it does not compile

(A **member** is anything declared directly in the class body: an instance variable, a constructor, or a method.)

Why check this when the program is compiled, instead of when it runs? Because a mistake caught by the compiler can never reach a user. The author of the class marks the rule once, and Java enforces it on every line of every other class.

KEY IDEA

A private variable, constructor, or method can be used only by code inside the class that declares it. A public one can be used by code in any class.

WORKED EXAMPLE One public variable, one private variable

This version of `Player` keeps `name` public but makes `points` private. `main` tries to set both.

PrivateOutside.java

```
1 public class PrivateOutside
2 {
3     public static void main(String[] args)
4     {
5         Player ana = new Player();
6         ana.name = "Ana";
7         ana.points = 12;
8     }
9 }
10
11 class Player
12 {
13     public String name;
14     private int points;
15
16     public Player()
17     {
18         name = "Nobody";
19         points = 0;
20     }
21
22     public void printCard()
23     {
24         System.out.println(name + ": " + points + " points");
25     }
26 }
```

- Line 6, `ana.name = "Ana";`, compiles: `name` is public, so the `main` method in `PrivateOutside` may use it.
- Line 7, `ana.points = 12;`, does not compile. `points` is private to `Player`, and line 7 is in a different class. Java reports (wording can vary):

```
PrivateOutside.java:7: error: points has private access in Player
```

Read the message literally: `points` is private, and only `Player` has access to it.

Private does not mean "unusable". It means "usable only from inside". Change one thing: instead of touching `points` from `main`, let `Player`'s own method use it.

WORKED EXAMPLE Private data used from inside the class

PrivateInside.java

```
1 public class PrivateInside
2 {
3     public static void main(String[] args)
4     {
5         Player ana = new Player();
6         ana.name = "Ana";
7         ana.printCard();
8     }
9 }
10
11 class Player
12 {
13     public String name;
14     private int points;
15
16     public Player()
17     {
18         name = "Nobody";
19         points = 0;
20     }
21
22     public void printCard()
23     {
24         System.out.println(name + ": " + points + " points");
25     }
26 }
```

This compiles and prints:

```
Ana: 0 points
```

Line 24 uses `points`, and line 24 is inside `printCard`, a method of `Player`. Code inside the declaring class may use private members, so this is allowed. `main` never names `points` at all: it calls the public method `printCard` (line 7), and that method, which belongs to the class, reads the private variable. The same private variable is off-limits on line 7 of `PrivateOutside` and fine on line 24 here. What decides is the class the line is written in.

WATCH OUT

"They are in the same file, so `main` can see it." In this chapter's examples, `main` and the private variable are only a few lines apart in one file, so it is natural to think `main` counts as "inside". It does not. `private` limits access to the declaring *class*, not to the file. `main` belongs to `PrivateOutside`, a different class from `Player`, so line 7 of `PrivateOutside.java` fails with `points has private access in Player`. The fix is to let `Player`'s own code do the work, as `printCard` does in the second example.

PAUSE AND TRY**Question 5** Multiple choice

A program has this class:

BadgeAllowed.java

```
1 public class BadgeAllowed
2 {
3     public static void main(String[] args)
4     {
5         Badge b = new Badge();
6         b.owner = "Kim";
7         b.printBadge();
8     }
9 }
10
11 class Badge
12 {
13     public String owner;
14     private int code;
15
16     public Badge()
17     {
18         owner = "Visitor";
19         code = 1000;
20     }
21
22     public void printBadge()
23     {
24         System.out.println(owner + " #" + code);
25     }
26 }
```

If one of the following lines is added to `main` after line 5, which one does **not** compile?

- A. `b.owner = "Kim";`
- B. `b.printBadge();`
- C. `b.code = 4410;`
- D. `Badge other = new Badge();`

Question 6 Short answer

A student says: "`printBadge` reads the private variable `code`. So `code` is not really private." Explain what is wrong with the student's reasoning.

3.3.4 Encapsulation Keeps Instance Variables Private

Now that you know what `private` does, when should you use it? Here is what can happen without it. This `Scoreboard` makes its scores `public`:

PublicScore.java

```
1 public class PublicScore
2 {
3     public static void main(String[] args)
4     {
5         Scoreboard field = new Scoreboard();
6         field.homeScore = -4;
7         field.printScores();
8     }
9 }
10
11 class Scoreboard
12 {
13     public int homeScore;
14     public int awayScore;
15
16     public Scoreboard()
17     {
18         homeScore = 0;
19         awayScore = 0;
20     }
21
22     public void printScores()
23     {
24         System.out.println("Home " + homeScore + ", Away " + awayScore);
25     }
26 }
```

It compiles and prints:

```
Home -4, Away 0
```

A score of `-4` is impossible in a real game, but nothing stopped line 6. Any code in any class could do the same, and if this app had a bug, every line in every class would be a suspect.

Make one change: mark the scores `private`.

PrivateScore.java

```
1 public class PrivateScore
2 {
3     public static void main(String[] args)
4     {
5         Scoreboard field = new Scoreboard();
6         field.homeScore = -4;
7         field.printScores();
8     }
9 }
10
11 class Scoreboard
12 {
13     private int homeScore;
14     private int awayScore;
15
16     public Scoreboard()
17     {
18         homeScore = 0;
19         awayScore = 0;
20     }
21
22     public void printScores()
23     {
24         System.out.println("Home " + homeScore + ", Away " + awayScore);
25     }
26 }
```

Now line 6 does not compile:

```
PrivateScore.java:6: error: homeScore has private access in Scoreboard
```

The impossible score never happens, because the only code allowed to change `homeScore` is the code inside `Scoreboard`. If a score is ever wrong, you only have to check one class.

This is the idea of **encapsulation** (also called data encapsulation): the details of how a class does its work are kept hidden inside the class, and other classes use the class only through the parts it makes public. A car is a good comparison. A driver uses a steering wheel, pedals, and a key, the public parts. The engine's wiring is sealed away, so a driver cannot accidentally connect the wrong wires, and the maker can redesign the engine without teaching anyone to drive again. The comparison stops at one point: a mechanic can still open the hood, but nothing in another Java class can ever reach a private member.

Encapsulation gives two benefits:

1. **Protection.** Outside code cannot put an object's data into a nonsense state. Only the class's own code changes its instance variables, so the class's author controls every change.
2. **Freedom to change.** Outside code depends only on the public parts, so the author can change private details later (rename a variable, store a value a different way) without breaking any other class. This is the same benefit you saw for procedural abstraction in Topic 3.1: users of a method need to know what it does, not how.

Because of this, the standard practice, and the rule for AP exam code, is: **mark instance variables private, unless the class's specification says otherwise.** A **specification** is the written

description of what a class must contain and do, such as the description at the top of an AP free-response question. If a specification explicitly says an instance variable is public, follow it; otherwise, write `private`.

KEY IDEA

Encapsulation keeps a class's data hidden inside the class. Mark every instance variable private unless the specification says otherwise, so only the class's own code can change it.

WORKED EXAMPLE Changing private data through public methods

If outside code cannot touch the scores, how does a score ever go up? The class offers public methods that change its own private data in allowed ways.

GoalCounter.java

```
1  public class GoalCounter
2  {
3      public static void main(String[] args)
4      {
5          Scoreboard field = new Scoreboard();
6          field.homeGoal();
7          field.awayGoal();
8          field.homeGoal();
9          field.printScores();
10     }
11 }
12
13 class Scoreboard
14 {
15     private int homeScore;
16     private int awayScore;
17
18     public Scoreboard()
19     {
20         homeScore = 0;
21         awayScore = 0;
22     }
23
24     public void homeGoal()
25     {
26         homeScore = homeScore + 1;
27     }
28
29     public void awayGoal()
30     {
31         awayScore = awayScore + 1;
32     }
33
34     public void printScores()
35     {
36         System.out.println("Home " + homeScore + ", Away " + awayScore);
37     }
38 }
```

Line	Statement	homeScore	awayScore
5	<code>new Scoreboard()</code>	0	0
6	<code>field.homeGoal();</code>	1	0
7	<code>field.awayGoal();</code>	1	1
8	<code>field.homeGoal();</code>	2	1

Each call runs inside `Scoreboard`. Line 6 calls `homeGoal`, so line 26 runs on `field`'s object: it reads that object's `homeScore`, adds 1, and stores the result. Line 9 prints:

```
Home 2, Away 1
```

`main` can still record goals, but only the way `Scoreboard` allows: one point at a time, upward. There is now no way for any other class to make a score negative. Topic 3.5 teaches how to write methods like these, including ones that take in values and give values back.

WATCH OUT

Leaving off the access modifier. Java does not force you to write `public` or `private`. If you write only `int homeScore;`, the class still compiles:

`NoModifier.java`

```

1  public class NoModifier
2  {
3      public static void main(String[] args)
4      {
5          Scoreboard field = new Scoreboard();
6          field.homeScore = -4;
7          field.printScores();
8      }
9  }
10
11 class Scoreboard
12 {
13     int homeScore;
14     int awayScore;
15
16     public Scoreboard()
17     {
18         homeScore = 0;
19         awayScore = 0;
20     }
21
22     public void printScores()
23     {
24         System.out.println("Home " + homeScore + ", Away " + awayScore);
25     }
26 }
```

It prints `Home -4, Away 0`. With no keyword, Java allows access from other classes in the same package (for simple projects, roughly the same folder), which includes a `main` in the

same file. So the impossible score is back, and the class is not encapsulated. Code with no access keyword does not appear in AP exam answers. Always write the keyword yourself:

```
private int homeScore;
```

PAUSE AND TRY

Question 7 Multiple choice

Which statement best explains why instance variables are usually declared `private`?

- A. Private variables use less memory than public variables.
- B. Private variables cannot be changed by any code, so their values stay the same.
- C. Private variables can be changed only by code inside their own class, so other classes cannot put an object into an invalid state.
- D. Private variables can be used only in `main`.

Question 8 Short answer

A free-response question says: "The `Ticket` class has an instance variable `seat` of type `int`." It says nothing about access. A second question says: "The `Gauge` class has a public instance variable `level` of type `double`." How should each variable be declared, and why?

3.3.5 Access for Constructors and Methods

`public` and `private` apply to more than variables. A class's constructors and methods also have access modifiers, and the right choice depends on who needs to use each one.

Constructors are always `public` in this course. The whole point of a constructor is to let other code build objects with `new`. A private constructor would block exactly that:

PrivateConstructor.java

```
1 public class PrivateConstructor
2 {
3     public static void main(String[] args)
4     {
5         Scoreboard field = new Scoreboard();
6         field.printScores();
7     }
8 }
9
10 class Scoreboard
11 {
12     private int homeScore;
13     private int awayScore;
14
15     private Scoreboard()
16     {
17         homeScore = 0;
18         awayScore = 0;
19     }
20 }
```

```

21     public void homeGoal()
22     {
23         homeScore = homeScore + 1;
24     }
25
26     public void printScores()
27     {
28         printDivider();
29         System.out.println("Home " + homeScore + ", Away " + awayScore);
30         printDivider();
31     }
32
33     private void printDivider()
34     {
35         System.out.println("-----");
36     }
37 }

```

Line 5 does not compile:

```
PrivateConstructor.java:5: error: Scoreboard() has private access in Scoreboard
```

`main` is outside `Scoreboard`, so it may not call a private constructor, and without a constructor call it cannot get a `Scoreboard` object at all.

Methods can be either. In Topic 3.1, behaviors were the actions of a class. Some actions are services for outside code: recording a goal, printing the scores. Those methods are `public`, because outside code must be able to call them. Other methods exist only to help the class's own code, such as a small step that several public methods share. Those are `private`: they are part of how the class works inside, which encapsulation keeps hidden.

KEY IDEA

Constructors are public, so any class can build objects. Methods that other classes need to call are public; helper methods used only by the class's own code are private.

WORKED EXAMPLE A private helper method

This `Scoreboard` prints a divider line above and below the scores. The divider is printed by its own method, `printDivider`, which is private.

`ScoreReport.java`

```

1  public class ScoreReport
2  {
3      public static void main(String[] args)
4      {
5          Scoreboard field = new Scoreboard();
6          field.homeGoal();
7          field.printScores();
8      }
9  }
10

```

```
11 class Scoreboard
12 {
13     private int homeScore;
14     private int awayScore;
15
16     public Scoreboard()
17     {
18         homeScore = 0;
19         awayScore = 0;
20     }
21
22     public void homeGoal()
23     {
24         homeScore = homeScore + 1;
25     }
26
27     public void printScores()
28     {
29         printDivider();
30         System.out.println("Home " + homeScore + ", Away " + awayScore);
31         printDivider();
32     }
33
34     private void printDivider()
35     {
36         System.out.println("-----");
37     }
38 }
```

Follow the calls:

1. Line 5 builds the object; line 6 calls `homeGoal`, so `homeScore` becomes 1.
2. Line 7 calls the public method `printScores`; Java jumps to line 29.
3. Line 29 calls `printDivider()`. Inside a class, a method can call another method of the same class by its name alone, and the call works on the same object. `printDivider` is private, but line 29 is inside `Scoreboard`, so the call is allowed. Line 36 prints the dashes.
4. Line 30 prints the scores, and line 31 calls `printDivider` again.

```
-----
Home 1, Away 0
-----
```

Change who makes the call. Now `main` tries to call the helper directly:

CallPrivateHelper.java

```
1 public class CallPrivateHelper
2 {
3     public static void main(String[] args)
4     {
5         Scoreboard field = new Scoreboard();
6         field.printDivider();
7     }
8 }
```

```

9
10 class Scoreboard
11 {
12     private int homeScore;
13     private int awayScore;
14
15     public Scoreboard()
16     {
17         homeScore = 0;
18         awayScore = 0;
19     }
20
21     public void homeGoal()
22     {
23         homeScore = homeScore + 1;
24     }
25
26     public void printScores()
27     {
28         printDivider();
29         System.out.println("Home " + homeScore + ", Away " + awayScore);
30         printDivider();
31     }
32
33     private void printDivider()
34     {
35         System.out.println("-----");
36     }
37 }

```

CallPrivateHelper.java:6: error: printDivider() has private access in Scoreboard

The method is identical; only the location of the call changed. Line 29 of `ScoreReport.java` is inside the class, so it may call the private method. Line 6 here is in another class, so it may not.

Why bother hiding a helper? If `printDivider` were public, other classes might start calling it. Then the author could not rename it, remove it, or change how it prints without breaking their code. Keeping it private means it is free to change, the same freedom private variables give.

WATCH OUT

Making a method private when outside code needs it. If a teammate marks `printScores` private "to be safe", the class still compiles, but no other class can ask a scoreboard for its scores anymore: every call like `field.printScores();` in `main` fails with `printScores() has private access in Scoreboard`. "Make everything private" applies to instance variables. For methods, ask who needs to call it: if other classes do, it must be public.

Here is a summary of the access rules this course uses:

Part of a class	Access in this course	Reason
Class	public	Other classes need to use it

Instance variable	<code>private</code> , unless the specification says otherwise	Encapsulation: only the class changes its data
Constructor	<code>public</code>	Other classes need to build objects with <code>new</code>
Method	<code>public</code> if other classes call it; <code>private</code> if only a helper	Public methods are services; helpers stay hidden

PAUSE AND TRY**Question 9** Debugging

This program is supposed to print the time-up message followed by `BEEP`, but it does not compile. Find the error and fix it with a change to `main` only.

TimerBug.java

```

1  public class TimerBug
2  {
3      public static void main(String[] args)
4      {
5          Timer quiz = new Timer();
6          quiz.beep();
7      }
8  }
9
10 class Timer
11 {
12     private int seconds;
13
14     public Timer()
15     {
16         seconds = 30;
17     }
18
19     public void finish()
20     {
21         System.out.println(seconds + " seconds are up");
22         beep();
23     }
24
25     private void beep()
26     {
27         System.out.println("BEEP");
28     }
29 }
```

Question 10 Multiple choice

Which constructor header for the `Timer` class follows this course's rules and compiles?

- A. `private Timer()`
- B. `public Timer()`
- C. `public timer()`
- D. `Timer public()`

Putting it together

This `Locker` class uses every idea in the chapter: private instance variables, a public constructor, public methods, and a private helper. Two lockers show that each object keeps its own data.

`LockerDemo.java`

```
1 public class LockerDemo
2 {
3     public static void main(String[] args)
4     {
5         Locker mine = new Locker();
6         Locker yours = new Locker();
7         mine.lock();
8         yours.lock();
9         yours.unlock();
10        System.out.print("Mine: ");
11        mine.printStatus();
12        System.out.print("Yours: ");
13        yours.printStatus();
14    }
15 }
16
17 class Locker
18 {
19     private boolean locked;
20     private int timesUsed;
21
22     public Locker()
23     {
24         locked = false;
25         timesUsed = 0;
26     }
27
28     public void lock()
29     {
30         locked = true;
31         countUse();
32     }
33
34     public void unlock()
35     {
36         locked = false;
37         countUse();
38     }
39
40     public void printStatus()
41     {
42         if (locked)
43         {
44             System.out.println("locked, uses: " + timesUsed);
45         }
46         else
47         {
48             System.out.println("open, uses: " + timesUsed);
```

```

49     }
50     }
51
52     private void countUse()
53     {
54         timesUsed++;
55     }
56 }

```

First, the layout of `Locker` (lines 17 to 56):

Lines	Part	Access
17	Class header (in its own file: <code>public class Locker</code>)	see 3.3.1
19–20	Instance variables <code>locked</code> and <code>timesUsed</code>	private
22–26	Constructor: starts every locker open and unused	public
28–38	Methods <code>lock</code> and <code>unlock</code> , called from <code>main</code>	public
40–50	Method <code>printStatus</code> , called from <code>main</code>	public
52–55	Helper <code>countUse</code> , called only by <code>lock</code> and <code>unlock</code>	private

Now trace `main`. Lines 5 and 6 build two `Locker` objects, each with its own `locked` (false) and `timesUsed` (0).

Line	Call	mine: locked, timesUsed	yours: locked, timesUsed
5–6	build both	false, 0	false, 0
7	<code>mine.lock()</code>	true, 1	false, 0
8	<code>yours.lock()</code>	true, 1	true, 1
9	<code>yours.unlock()</code>	true, 1	false, 2

Each `lock` or `unlock` call sets `locked` and then calls the private helper `countUse` (line 31 or 37), which adds 1 to that same object's `timesUsed` (line 54). `mine` was used once; `yours` was used twice and ended open. Lines 10 to 13 print a label and then call `printStatus`, whose `if` (Topic 2.3) picks the message:

```

Mine: locked, uses: 1
Yours: open, uses: 2

```

Finally, check which lines `main` is allowed to contain:

Line in <code>main</code>	Compiles?	Why
<code>mine.lock();</code>	Yes	<code>lock</code> is public
<code>mine.locked = true;</code>	No	<code>locked</code> is private: <code>locked</code> has private access in <code>Locker</code>
<code>mine.countUse();</code>	No	<code>countUse</code> is private: <code>countUse()</code> has private access in <code>Locker</code>
<code>Locker extra = new Locker();</code>	Yes	The constructor is public

Because `locked` and `timesUsed` are private, the only way to change them is through `lock` and `unlock`, and every use is counted. No outside code can lock a locker without counting it, or set `timesUsed` back to 0.

Vocabulary

Term	Definition	Example
class	A blueprint for objects, declared with the keyword <code>class</code> ; in this course always <code>public</code>	<code>public class Locker</code>
class header	The first line of a class: access modifier, <code>class</code> , and the class name	<code>public class Scoreboard</code>
instance variable	A variable declared in the class body, outside any constructor or method; each object has its own copy	<code>private int homeScore;</code>
constructor	The code, named like the class, that runs when <code>new</code> builds an object; in this course always <code>public</code>	<code>public Scoreboard()</code>
method	A named behavior of the class's objects	<code>public void homeGoal()</code>
member	Anything declared directly in the class body: an instance variable, a constructor, or a method	<code>homeScore</code> , <code>printScores</code>
access modifier	A keyword that decides which code may use a class or member	<code>public</code> , <code>private</code>
public	Usable by code in any class	<code>public void printScores()</code>
private	Usable only by code inside the class that declares it	<code>private void printDivider()</code>
encapsulation	Keeping a class's details hidden inside it, so other classes use only its public parts	Private scores, public <code>homeGoal</code>
specification	The written description of what a class must contain and do	"Ticket has a public seat"

Chapter summary

- A class has a header (`public class Name`) and a body that holds its instance variables, then its constructors, then its methods. In this course every class is `public`. A file can hold only one `public` class, named like the file, so real projects put each class in its own file.
- Instance variables belong to the object: every object built with `new` gets its own copy, so changing one object's variable never changes another object's. Two variables share data only when they refer to the same object.
- `private` limits a member to code inside the declaring class; `public` allows code in any class. The deciding question is which class a line of code is written in, not which file.
- Encapsulation keeps a class's data hidden inside the class. Instance variables are `private` unless the specification says otherwise, so only the class's own code can change them.
- Constructors are always `public` so other classes can build objects. Methods other classes need are `public`; helper methods used only inside the class are `private`.

Review questions

Question 11 Tracing

This program uses the `Scoreboard` class from 3.3.4 (private `scores`, public `homeGoal`, `awayGoal`, and `printScores`). What does it print?

`TwoBoards.java`

```
1 public class TwoBoards
2 {
3     public static void main(String[] args)
4     {
5         Scoreboard field = new Scoreboard();
6         field.homeGoal();
7         field.homeGoal();
8         Scoreboard gym = new Scoreboard();
9         gym.awayGoal();
10        field.printScores();
11        gym.printScores();
12    }
13 }
14
15 class Scoreboard
16 {
17     private int homeScore;
18     private int awayScore;
19
20     public Scoreboard()
21     {
22         homeScore = 0;
23         awayScore = 0;
24     }
25
26     public void homeGoal()
27     {
28         homeScore = homeScore + 1;
29     }
30
31     public void awayGoal()
32     {
33         awayScore = awayScore + 1;
34     }
35
36     public void printScores()
37     {
38         System.out.println("Home " + homeScore + ", Away " + awayScore);
39     }
40 }
```

Question 12 Multiple choice

Consider the `Timer` class from Question 9 (private `seconds`, public constructor, public `finish`, private `beep`). Which of these statements, written in the `main` method of a different class, compile?

```
1 Timer t = new Timer();    // I
2 t.seconds = 60;           // II
3 t.finish();               // III
```

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

Question 13 Debugging

A student wants `Score` to stay hidden from other code, so they mark the class itself `private`. The program does not compile. Find the error and give the corrected line for this one-file program.

PrivateClassHeader.java

```
1 public class PrivateClassHeader
2 {
3     public static void main(String[] args)
4     {
5         Score game = new Score();
6         game.printScore();
7     }
8 }
9
10 private class Score
11 {
12     private int points;
13
14     public Score()
15     {
16         points = 0;
17     }
18
19     public void printScore()
20     {
21         System.out.println(points);
22     }
23 }
```

Question 14 Short answer

In `LockerDemo`, a teammate suggests making `timesUsed` public "so `main` can reset a locker's count when the school year starts." Using the idea of encapsulation, explain what risk this creates.

Question 15 Multiple choice

A class has a private instance variable `rating`, a public method `printRating` that prints it, and a private method `checkRange` that `printRating` calls. Which statement is true?

- A. Every method of this class may use `rating` and call `checkRange`, but other classes may call only `printRating`.
- B. `printRating` may not use `rating`, because `rating` is private and `printRating` is public.
- C. Other classes may use `rating` as long as they are in the same file as this class.
- D. `checkRange` may be called from another class's `main`, because `printRating` calls it.

Reflection

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

☐ I can find the class header, instance variables, constructor, and methods in a class. Reread: 3.3.1

The Parts of a Class

☐ I can explain why a one-file example marks only one class public, and where each class goes in a real project. Reread: Why this program's Scoreboard is not marked public

☐ I can trace code with several objects and explain why each has its own instance variables.

Reread: 3.3.2 Each Object Has Its Own Instance Variables

☐ I can decide whether a line of code may use a public or private member. Reread: 3.3.3 The public and private Keywords

☐ I can explain encapsulation and why instance variables are private unless a specification says otherwise. Reread: 3.3.4 Encapsulation Keeps Instance Variables Private

☐ I can explain why constructors are public and choose public or private for a method. Reread: 3.3.5 Access for Constructors and Methods

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. `isOn` and `brightness` are declared inside the class but outside any constructor or method, so they are the two instance variables. `Lamp` has the class's name and no `void`, so it is the constructor. `switchOn` has its own name and `void`, so it is a method.

- **A** is wrong for two reasons: `brightness` is also an instance variable, and `Lamp` is not a method. A block named after the class with no return type is the constructor.
- **C** is wrong because `switchOn` is not named after the class and has `void` in its header, so it is a method, not a constructor.
- **D** is wrong because the lines inside the constructor (`isOn = false;`) are assignments that give the existing variables values. They do not declare new variables; a declaration starts with a type, such as `private boolean isOn;`.

Reread 3.3.1 *The Parts of a Class* if you chose differently.

Question 2

Model answer: A Java file can hold only one `public` class, and that class must have the same name as the file. `LampTest` is correctly in `LampTest.java`, but `Lamp` is also `public`, and Java expects a `public` class named `Lamp` to be in a file named `Lamp.java`. A real project moves `Lamp` into its own file, `Lamp.java`, where it stays `public`. (In a one-file demonstration, removing `public` from `Lamp` would also compile.)

A complete answer must:

- state the rule: one `public` class per file, named the same as the file, and
- give the fix of putting `Lamp` in `Lamp.java`.

Reread *Why this program's Scoreboard is not marked public* if your answer missed the file-name rule.

Question 3

Output:

```
1
14
14
```

Lines 5 and 6 build two objects. Call them object P (referred to by `p`) and object Q (referred to by `q`). Line 7 copies `q`'s reference into `r`, so `r` also refers to object Q.

Line	Statement	P's points	Q's points
5–7	build objects, copy reference	0	0
8	<code>p.points = 5;</code>	5	0
9	<code>q.points = 9;</code>	5	9
10	<code>r.points = r.points + p.points;</code>	5	9 + 5 = 14

11	p.points = 1;	1	14
----	---------------	---	----

Line 10 changes Q's `points`, because `r` refers to object Q. Line 11 changes only P. `q.points` and `r.points` are the same variable, so both print 14.

Reread 3.3.2 *Each Object Has Its Own Instance Variables* if you got a different result.

Question 4

B. Each object has one copy of each instance variable, so count the objects. `new` appears three times, so there are three `Player` objects and three `points` variables. `d = b` copies a reference; it does not build an object.

- **A** is wrong because it treats `points` as belonging to the class. Instance variables belong to each object.
- **C** is wrong because it counts variables (`a`, `b`, `c`, `d`) instead of objects. `d` and `b` refer to the same object.
- **D** is wrong; there is no way to reach five, since only three objects exist.

Reread 3.3.2 *Each Object Has Its Own Instance Variables* if you chose differently.

Question 5

C. `code` is declared `private` in `Badge`, and `main` is in a different class, `BadgeAllowed`. Java reports `code` has private access in `Badge`.

- **A** compiles: `owner` is `public`, so any class may use it. (Lines 6 and 7 of the program already do this.)
- **B** compiles: `printBadge` is a `public` method. It uses the `private code` on line 24, but that line is inside `Badge`, which is allowed. The program prints `Kim #1000`.
- **D** compiles: the constructor `Badge()` is `public`, so `main` may build another object.

Reread 3.3.3 *The public and private Keywords* if you chose differently.

Question 6

Model answer: `private` does not mean that no code may use the variable. It means only code inside the declaring class may use it. `printBadge` is a method of `Badge`, the class that declares `code`, so it is allowed to read `code`. Code in other classes still cannot name `code` directly; they can only call public methods such as `printBadge`, which decide what to do with it.

A complete answer must:

- state that private access is limited to the declaring class, not forbidden everywhere, and
- note that `printBadge` is inside `Badge`, while outside code still cannot reach `code` directly.

Reread 3.3.3 *The public and private Keywords* if your explanation differed.

Question 7

C. This is encapsulation: only the class's own constructors and methods may use its private instance variables, so the class controls every change.

- **A** is wrong because access modifiers have nothing to do with memory; a `private int` and a `public int` take the same space.
- **B** is wrong because code inside the class can change a private variable, as `homeGoal` does. `private` limits *who* may change it, not *whether* it can change.

- **D** is backwards. `main` in another class is exactly the kind of outside code that cannot use a private variable.

Reread 3.3.4 *Encapsulation Keeps Instance Variables Private* if you chose differently.

Question 8

Model answer: `seat` should be declared `private int seat;`. The specification says nothing about access, and the standard practice for encapsulation is to make instance variables private. `level` should be declared `public double level;`, because this specification explicitly states that it is public, and a specification's stated requirement overrides the usual practice.

A complete answer must:

- give `private` for `seat` with the encapsulation reason, and
- give `public` for `level` because the specification says so.

Reread 3.3.4 *Encapsulation Keeps Instance Variables Private* if your answer differed.

Question 9

Error: line 6 calls `quiz.beep()`, but `beep` is `private` in `Timer` (line 25), and `main` is in a different class. Java reports `beep()` has private access in `Timer`.

Fix: call the public method that uses the helper:

```
1      Timer quiz = new Timer();
2      quiz.finish();
```

`finish` is public, so `main` may call it. Inside the class, `finish` prints 30 seconds are up and then calls `beep()`, which is allowed because that call is inside `Timer`. The program prints:

```
30 seconds are up
BEEP
```

Making `beep` public would also compile, but it breaks encapsulation: the helper is a detail of how `finish` works, and outside code was never meant to call it.

Reread 3.3.5 *Access for Constructors and Methods* if you tried a different fix.

Question 10

B. A constructor is named exactly like its class and, in this course, is always `public`, with the access modifier written first.

- **A** compiles, but a private constructor means no other class can build a `Timer` with `new`, so it breaks this course's rule that constructors are public.
- **C** does not compile. Java names are case-sensitive, so `timer` is not the class name `Timer`. Java treats it as a method with no return type and reports `invalid method declaration; return type required.`
- **D** does not compile. The access modifier must come before the name, so Java reports `<identifier> expected.`

Reread 3.3.5 *Access for Constructors and Methods* if you chose differently.

Question 11

Output:

```
Home 2, Away 0
Home 0, Away 1
```

There are two `Scoreboard` objects, and each has its own `homeScore` and `awayScore`.

Line	Call	field: home, away	gym: home, away
5	build field	0, 0	(not built yet)
6	field.homeGoal()	1, 0	
7	field.homeGoal()	2, 0	
8	build gym	2, 0	0, 0
9	gym.awayGoal()	2, 0	0, 1

`gym` was built after `field`'s goals, but that does not matter: its constructor gives its own variables their starting values, and `field`'s goals live in `field`'s object.

Reread 3.3.2 *Each Object Has Its Own Instance Variables* if your output differed.

Question 12

B. Statement I calls the public constructor, and statement III calls the public method `finish`; both are allowed from any class. Statement II names the private instance variable `seconds` from outside `Timer`, so Java reports `seconds` has private access in `Timer`.

- **A** is wrong because it leaves out III. `finish` is public; the fact that it uses the private `seconds` and `beep` inside the class does not stop outside code from calling it.
- **C** is wrong because it includes II and leaves out I. The constructor is public, and `seconds` is private.
- **D** is wrong because it includes II.

Reread 3.3.3 *The public and private Keywords* and 3.3.5 *Access for Constructors and Methods* if you chose differently.

Question 13

Error: line 10, `private class Score`. A class written on its own like this may not be `private`; Java reports `modifier private not allowed here` (wording can vary). In this course, classes are always `public`, because the point of a class is to let other code build and use its objects.

Fix for this one-file program:

```
1 class Score
```

Remove `private`. Since `PrivateClassHeader` is already the `public` class in this file, `Score` cannot be `public` here; in a real project, `Score` would go in its own file, `Score.java`, as `public class Score`. With the fix, the program prints 0.

What the student actually wanted, hiding the data, is done by the instance variable: `points` is already `private`, so encapsulation does not need the class itself to be `private`.

Reread 3.3.1 *The Parts of a Class* if you tried a different fix.

Question 14

Model answer: If `timesUsed` were public, any line in any class could set it to any value, not only to 0 for a reset: it could be set to -3, or set back to 0 by mistake in the middle of the year. The class could no longer guarantee that the count matches the real number of `lock` and `unlock` calls, and if the count were wrong, every class would be a suspect instead of only `Locker`. Keeping it private means only `Locker`'s own code changes it. If a reset is truly needed, the class can offer a public method for it, so the change still happens in a way the class controls.

A complete answer must:

- say that public access lets any outside code set the variable to any value, and
- connect private access to the class controlling every change to its own data.

Reread 3.3.4 *Encapsulation Keeps Instance Variables Private* if your answer missed either point.

Question 15

A. Private members are usable by any code inside the declaring class, whether that code is in a public method or a private one. Outside the class, only the public members (here `printRating`, plus the constructor) are usable.

- **B** is wrong because the access modifier of the method doing the using does not matter. What matters is that `printRating` is inside the class, just as `printCard` used the private `points` in 3.3.3.
- **C** is wrong because private access is limited to the class, not the file.
- **D** is wrong because a call from inside the class does not make the helper available outside it; a call to `checkRange` from `main` would not compile, just as the call to `printDivider` from `main` did not.

Reread 3.3.3 *The public and private Keywords* and 3.3.5 *Access for Constructors and Methods* if you chose differently.