

3.6 Methods: Passing and Returning

References of an Object

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

By the end you will be able to

- Explain what a parameter holds when the argument is an object, and why no new object is made.
- Predict when a method or constructor changes the caller's object and when it does not.
- Explain why a method should leave the objects passed to it unchanged unless its job is to change them.
- Trace methods that return an object reference, and explain why the caller gets the same object, not a copy.
- Decide whether a method may use the private instance variables or private methods of an object passed to it.

What to remember from earlier

- A primitive argument is copied into the parameter, so changing the parameter never changes the caller's variable. ([Topic 3.5: Methods: How to Write Them](#), [3.5.6 Parameters Receive Copies](#))
- Accessors report an instance variable and mutators change one. ([Topic 3.5: Methods: How to Write Them](#))
- Assigning one reference variable to another copies only the address, so both are aliases of one object. ([Topic 1.12: Objects: Instances of Classes](#), [1.12.5 Assigning and Copying References](#))
- `==` on two references is true only when they refer to the very same object. ([Topic 2.6: Comparing Boolean Expressions](#), [2.6.4 Aliases and Reference Comparison](#))
- A String can never be changed after it is created. ([Topic 1.15: String Manipulation](#), [1.15.7 Strings Are Immutable](#))
- Private instance variables and methods can be used only inside their own class. ([Topic 3.3: Anatomy of a Class](#), [3.3.3 The public and private Keywords](#))

3.6.1 Passing an Object Reference

A game app keeps a `Player` object for each player, with a name and a point total. A `Scorekeeper` object awards bonuses. Its method needs to receive a whole player, not just a number, and the bonus has to stick: after the method finishes, the player in `main` should have the extra points. In Topic 3.5 every parameter you wrote was a primitive, such as an `int`, and changing an `int` parameter never changed anything outside the method. So what happens when the argument is an object?

Start from two things you already know. First, from Topic 3.5: a parameter is initialized with a **copy** of its argument. Second, from Topic 1.13: a variable such as `Player star` does not hold the player object itself. It holds a **reference**, the address of where the object lives in memory.

Put those together and the answer follows. When the argument is a reference variable, what gets copied into the parameter is the **reference**, the address. The object is not copied. There is still exactly one `Player` object, and now two variables lead to it: the caller's variable and the method's parameter. In the words of Topic 2.6, the parameter becomes an **alias** of the caller's object for as long as the method runs.

Think of handing a friend a slip of paper with your home address copied onto it. Your friend has their own slip, but there is still only one house. If they go there and paint the front door, you will see the new color when you come home. (The analogy stops at the slip itself: if your friend scribbles a different address on *their* slip, your slip is unchanged. That case is the subject of 3.6.2.)

Because the method can reach the real object, it can call that object's mutators and change its state. An object whose state can be changed after it is created, such as a `Player` with an `addPoints` mutator, is called a **mutable** object. A `String` is not mutable, as you saw in Topic 1.15.

KEY IDEA

When the argument is an object, the parameter gets a copy of the reference, not a copy of the object. The parameter and the caller's variable refer to the same object, so a method can use the parameter to change a mutable object's state.

Most programs in this chapter use a `Player` class like the one in the next example: private `name` and `points`, a constructor, two accessors, and the mutator `addPoints`. In your own projects each class is `public` and goes in its own file. To keep each example runnable as a single file, the extra classes here leave off `public`, because Java allows only one `public` class per file. Once you have read `Player` here, you can skip it at the bottom of later examples.

WORKED EXAMPLE A bonus that sticks

Watch which object line 16 changes, and what line 8 prints afterward.

BonusByReference.java

```
1 public class BonusByReference
2 {
3     public static void main(String[] args)
4     {
5         Player star = new Player("Ada", 10);
6         Scorekeeper keeper = new Scorekeeper();
7         keeper.awardBonus(star);
8         System.out.println("Back in main: " + star.getPoints());
9     }
10 }
11
12 class Scorekeeper
13 {
14     public void awardBonus(Player p)
15     {
16         p.addPoints(5);
17         System.out.println("Inside awardBonus: " + p.getPoints());
18     }
19 }
```

```

20
21 class Player
22 {
23     private String name;
24     private int points;
25
26     public Player(String playerName, int startPoints)
27     {
28         name = playerName;
29         points = startPoints;
30     }
31
32     public String getName()
33     {
34         return name;
35     }
36
37     public int getPoints()
38     {
39         return points;
40     }
41
42     public void addPoints(int amount)
43     {
44         points = points + amount;
45     }
46 }

```

Line 5 builds one `Player` object. Call its address "address A." `star` holds that address.

Line 7 calls `awardBonus(star)`. The parameter `p` is initialized with a copy of what `star` holds, which is "address A." No `new` appears anywhere in the call, so no second player is built. While the method runs, memory looks like this:

Variable	Holds	Object at that address
<code>star</code> (in <code>main</code>)	reference to address A	<code>Player</code> "Ada", 10 points
<code>p</code> (in <code>awardBonus</code>)	reference to address A	(the same object)

Line 16 follows `p` to address A and calls `addPoints(5)`, so that one object now has 15 points.

Line 17 prints `Inside awardBonus: 15`.

The method ends and `p` disappears, but the object at address A is still there with its 15 points. Back in `main`, line 8 follows `star` to that same object:

Variable	Holds	Object at that address
<code>star</code> (in <code>main</code>)	reference to address A	<code>Player</code> "Ada", 15 points

```

Inside awardBonus: 15
Back in main: 15

```

Now change one thing: pass a primitive instead of an object. This version gives `awardBonus` an `int` parameter and adds 5 to it.

BonusByValue.java

```

1 public class BonusByValue
2 {
3     public static void main(String[] args)
4     {
5         int adaScore = 10;
6         Scorekeeper keeper = new Scorekeeper();
7         keeper.awardBonus(adaScore);
8         System.out.println("Back in main: " + adaScore);
9     }
10 }
11
12 class Scorekeeper
13 {
14     public void awardBonus(int score)
15     {
16         score = score + 5;
17         System.out.println("Inside awardBonus: " + score);
18     }
19 }

```

```

Inside awardBonus: 15
Back in main: 10

```

Line 7 copies the **value** 10 into `score`. Line 16 changes `score` to 15, but `score` is a separate box from `adaScore`, and there is no object behind either one. `adaScore` is still 10. Side by side:

Argument type	What the parameter gets	Can the method change what the caller sees?
Primitive (int)	A copy of the value	No. The caller's variable is separate.
Object (Player)	A copy of the reference (address)	Yes, by calling a mutator on the shared object.

In both cases Java copies what the argument variable holds. The difference is *what* it holds: a number in one case, an address in the other.

WATCH OUT

Thinking the method gets its own copy of the object. Many students picture `awardBonus(star)` as handing the method a photocopy of the player, so they predict `Back in main: 10` for `BonusByReference`. That would be true only if the call made a new object, and making an object always takes `new` (Topic 1.13). The call copies an address, so the method and `main` work on one object, and the program prints `Back in main: 15`. When you trace, ask "which object does this variable lead to?" rather than "which variable is this?"

PAUSE AND TRY**Question 1** Tracing

Write the exact output.

DoubleUp.java

```
1 public class DoubleUp
2 {
3     public static void main(String[] args)
4     {
5         Player grace = new Player("Grace", 4);
6         Coach coach = new Coach();
7         coach.doubleUp(grace);
8         coach.doubleUp(grace);
9         System.out.println(grace.getPoints());
10    }
11 }
12
13 class Coach
14 {
15     public void doubleUp(Player p)
16     {
17         p.addPoints(p.getPoints());
18     }
19 }
20
21 class Player
22 {
23     private String name;
24     private int points;
25
26     public Player(String playerName, int startPoints)
27     {
28         name = playerName;
29         points = startPoints;
30     }
31
32     public String getName()
33     {
34         return name;
35     }
36
37     public int getPoints()
38     {
39         return points;
40     }
41
42     public void addPoints(int amount)
43     {
44         points = points + amount;
45     }
46 }
```

Question 2 Multiple choice

In `BonusByReference`, what does the parameter `p` hold while `awardBonus` runs?

- A. A new `Player` object with the same name and points as `star`'s object
- B. A copy of the reference stored in `star`, so `p` and `star` lead to the same object

- C. A copy of the points value, 10
- D. The variable `star` itself, so assigning to `p` would also change `star`

3.6.2 What a Method Cannot Change

If a method can change the caller's object, can it also make the caller's variable refer to a **different** object? Suppose a `reset` method tries to replace a player with a brand-new rookie. Does the caller end up with the rookie?

The address slip from 3.6.1 answers it. The parameter is the method's own slip. Following the slip to the house and painting the door changes the house everyone shares. Erasing the slip and writing a new address on it changes only that slip. The caller's slip still has the old address.

In Java terms, there are two very different things a method can do with an object parameter:

Statement in the method	What it changes	Does the caller see it?
<code>p.addPoints(5);</code>	The object <code>p</code> refers to	Yes, the caller refers to that object too
<code>p = new Player("Rookie", 0);</code>	The parameter <code>p</code> (which address it holds)	No, the caller's variable still holds the old address

The dot in `p.addPoints` means "go to the object and act on it." An assignment `p = ...` means "put a different address in `p`." Only the first one reaches the shared object.

KEY IDEA

Assigning a new reference to a parameter changes only the parameter. The caller's variable keeps referring to the same object it did before the call.

WORKED EXAMPLE Replacing the parameter

Line 16 assigns a new object to `p`. Watch what `star` refers to before and after.

ReplaceParameter.java

```

1 public class ReplaceParameter
2 {
3     public static void main(String[] args)
4     {
5         Player star = new Player("Ada", 10);
6         Scorekeeper keeper = new Scorekeeper();
7         keeper.reset(star);
8         System.out.println("Back in main: " + star.getName() + " " +
9         star.getPoints());
10    }
11 }
12 class Scorekeeper
13 {
14     public void reset(Player p)

```

```

15     {
16         p = new Player("Rookie", 0);
17         System.out.println("Inside reset: " + p.getName() + " " + p.getPoints());
18     }
19 }
20
21 class Player
22 {
23     private String name;
24     private int points;
25
26     public Player(String playerName, int startPoints)
27     {
28         name = playerName;
29         points = startPoints;
30     }
31
32     public String getName()
33     {
34         return name;
35     }
36
37     public int getPoints()
38     {
39         return points;
40     }
41
42     public void addPoints(int amount)
43     {
44         points = points + amount;
45     }
46 }

```

Line 7 copies `star`'s reference, "address A," into `p`. Then line 16 runs. The right side, `new Player("Rookie", 0)`, builds a second object at "address B" and the `=` stores address B in `p`:

Variable	Holds	Object at that address
<code>star</code> (in main)	reference to address A	Player "Ada", 10 points
<code>p</code> (in reset)	reference to address B	Player "Rookie", 0 points

Line 17 follows `p` to address B and prints `Inside reset: Rookie 0`. When the method ends, `p` disappears. Nothing holds address B anymore, so the rookie is lost. `star` never stopped holding address A, so line 8 prints Ada's unchanged data.

```

Inside reset: Rookie 0
Back in main: Ada 10

```

The same rule explains what happens with a `String` parameter. A `String` is immutable (Topic 1.15): it has no mutators, so a method cannot change the text of the object it receives. All it can do is build a new string and store that in its parameter.

CheerString.java

```
1 public class CheerString
2 {
3     public static void main(String[] args)
4     {
5         String cheer = "Go team";
6         Announcer announcer = new Announcer();
7         announcer.shout(cheer);
8         System.out.println("Back in main: " + cheer);
9     }
10 }
11
12 class Announcer
13 {
14     public void shout(String text)
15     {
16         text = text + "!!!";
17         System.out.println("Inside shout: " + text);
18     }
19 }
```

```
Inside shout: Go team!!!
Back in main: Go team
```

Line 16 builds a new `String`, "Go team!!!", and moves `text` to it. That is an assignment to the parameter, like line 16 of `ReplaceParameter`, so `cheer` in `main` still refers to the original "Go team". A method can change the state of a mutable object passed to it, but it can never change a `String` argument.

WATCH OUT

Trying to swap two objects by swapping parameters. This method is meant to make `first` and `second` trade players:

SwapPlayersWrong.java

```
1 public class SwapPlayersWrong
2 {
3     public static void main(String[] args)
4     {
5         Player first = new Player("Ada", 10);
6         Player second = new Player("Grace", 20);
7         Scorekeeper keeper = new Scorekeeper();
8         keeper.swap(first, second);
9         System.out.println("first: " + first.getName());
10        System.out.println("second: " + second.getName());
11    }
12 }
13
14 class Scorekeeper
15 {
16     public void swap(Player a, Player b)
17     {
```

```
18     Player temp = a;
19     a = b;
20     b = temp;
21 }
22 }
23
24 class Player
25 {
26     private String name;
27     private int points;
28
29     public Player(String playerName, int startPoints)
30     {
31         name = playerName;
32         points = startPoints;
33     }
34
35     public String getName()
36     {
37         return name;
38     }
39
40     public int getPoints()
41     {
42         return points;
43     }
44
45     public void addPoints(int amount)
46     {
47         points = points + amount;
48     }
49 }
```

```
first: Ada
second: Grace
```

Nothing traded. Lines 18 to 20 swap the addresses held by `a` and `b`, which are the method's own copies. `first` and `second` in `main` were never assigned, so they still refer to `Ada` and `Grace`. Students write this because the three-step swap with `temp` works on ordinary variables in one place. It cannot work across a method call, because a method has no way to put a new address into the caller's variables. A method only affects the caller through the objects it changes with mutators, or through a value it returns (3.6.4).

PAUSE AND TRY

Question 3 Tracing

Write the exact output.

MixedUpdate.java

```
1 public class MixedUpdate
2 {
3     public static void main(String[] args)
4     {
5         Player kai = new Player("Kai", 10);
6         Scorekeeper keeper = new Scorekeeper();
7         keeper.update(kai);
8         System.out.println(kai.getName() + " " + kai.getPoints());
9     }
10 }
11
12 class Scorekeeper
13 {
14     public void update(Player p)
15     {
16         p.addPoints(3);
17         p = new Player("Temp", 100);
18         p.addPoints(3);
19     }
20 }
21
22 class Player
23 {
24     private String name;
25     private int points;
26
27     public Player(String playerName, int startPoints)
28     {
29         name = playerName;
30         points = startPoints;
31     }
32
33     public String getName()
34     {
35         return name;
36     }
37
38     public int getPoints()
39     {
40         return points;
41     }
42
43     public void addPoints(int amount)
44     {
45         points = points + amount;
46     }
47 }
```

Question 4 Short answer

A classmate wants a method that adds an exclamation point to the caller's `String` so that the caller's variable shows the new text after the call. Explain why no method with a `String` parameter can do this.

3.6.3 Leave Parameters Unchanged Unless Asked

Being able to change a caller's object is powerful, and it is also a trap. Someone who calls a method named `previewBonus` expects to see a number, not to have their player's score changed. If a method quietly changes an object passed to it, the caller has no hint from the call itself, and the bug shows up later somewhere else.

A method's **specification** is the description of what it is supposed to do, often written as a comment just above its header. Callers rely on the specification instead of reading the method's code. Recall from Topic 3.5 that accessors report information while mutators change it. The good-practice rule follows: a method or constructor **should not modify a mutable object passed as a parameter unless its specification says to**. If the job is to report, compute, or record something, the method should read the object through its accessors and leave it alone. If the job is to change the object, as with `awardBonus`, changing it is exactly right.

A changed parameter object that the caller did not ask for is called a **side effect**. It is legal Java, so the compiler never complains. Only careful reading of the specification catches it.

KEY IDEA

Do not change an object passed to a method or constructor unless the specification requires that change. Read it with accessors instead.

WATCH OUT

A "preview" that changes the player. The comment on line 15 is the specification: report what the total *would be*.

PreviewChanges.java

```
1 public class PreviewChanges
2 {
3     public static void main(String[] args)
4     {
5         Player ada = new Player("Ada", 10);
6         Scorekeeper keeper = new Scorekeeper();
7         System.out.println(keeper.previewBonus(ada, 5));
8         System.out.println(keeper.previewBonus(ada, 5));
9         System.out.println(ada.getPoints());
10    }
11 }
12
13 class Scorekeeper
```

```
14 {
15     // Returns what p's total would be if p earned bonus more points.
16     public int previewBonus(Player p, int bonus)
17     {
18         p.addPoints(bonus);
19         return p.getPoints();
20     }
21 }
22
23 class Player
24 {
25     private String name;
26     private int points;
27
28     public Player(String playerName, int startPoints)
29     {
30         name = playerName;
31         points = startPoints;
32     }
33
34     public String getName()
35     {
36         return name;
37     }
38
39     public int getPoints()
40     {
41         return points;
42     }
43
44     public void addPoints(int amount)
45     {
46         points = points + amount;
47     }
48 }
```

```
15
20
20
```

The first call prints 15, which looks right. But line 18 actually added 5 to Ada's object, so the second call starts from 15 and prints 20, and Ada now really has 20 points. The method does more than its specification promises. Students write it this way because "add the bonus, then get the total" is how they would do it on paper. The fix computes the answer without touching the object:

```
1 return p.getPoints() + bonus;
```

With that single line as the method body, the program prints 15, 15, and 10: the preview is the same every time, and Ada keeps her 10 points.

Constructors follow the same rule, because a constructor's parameters are initialized the same way. This `Team` constructor is specified only to record the captain's name, but line 19 also changes the `Player` it was given:

`WelcomeBonus.java`

```
1 public class WelcomeBonus
2 {
3     public static void main(String[] args)
4     {
5         Player ada = new Player("Ada", 10);
6         Team robots = new Team(ada);
7         System.out.println(robots.getCaptainName() + " has " + ada.getPoints());
8     }
9 }
10
11 class Team
12 {
13     private String captainName;
14
15     // Records the name of the team's captain.
16     public Team(Player captain)
17     {
18         captainName = captain.getName();
19         captain.addPoints(10);
20     }
21
22     public String getCaptainName()
23     {
24         return captainName;
25     }
26 }
27
28 class Player
29 {
30     private String name;
31     private int points;
32
33     public Player(String playerName, int startPoints)
34     {
35         name = playerName;
36         points = startPoints;
37     }
38
39     public String getName()
40     {
41         return name;
42     }
43
44     public int getPoints()
45     {
46         return points;
47     }
48
49     public void addPoints(int amount)
50     {
```

```
51         points = points + amount;
52     }
53 }
```

Ada has 20

Line 6 passes `ada`'s reference to the constructor, so `captain` and `ada` refer to the same object. Line 18 reads the name with an accessor, which is fine. Line 19 calls a mutator on the shared object, so Ada's points jump from 10 to 20 just because a team was created. Deleting line 19 makes the constructor match its specification, and the program would print `Ada has 10`.

PAUSE AND TRY

Question 5 Debugging

The specification on line 15 says the method returns the total of the two players' points. The returned total is correct, but the method has a side effect. Identify the problem and write a corrected method body.

CombinedBug.java

```
1  public class CombinedBug
2  {
3      public static void main(String[] args)
4      {
5          Player lee = new Player("Lee", 8);
6          Player mia = new Player("Mia", 6);
7          Scorekeeper keeper = new Scorekeeper();
8          System.out.println(keeper.combinedPoints(lee, mia));
9          System.out.println(lee.getPoints());
10     }
11 }
12
13 class Scorekeeper
14 {
15     // Returns the total points of players a and b.
16     public int combinedPoints(Player a, Player b)
17     {
18         a.addPoints(b.getPoints());
19         return a.getPoints();
20     }
21 }
22
23 class Player
24 {
25     private String name;
26     private int points;
27
28     public Player(String playerName, int startPoints)
29     {
30         name = playerName;
31         points = startPoints;
32     }
```

```
33
34     public String getName()
35     {
36         return name;
37     }
38
39     public int getPoints()
40     {
41         return points;
42     }
43
44     public void addPoints(int amount)
45     {
46         points = points + amount;
47     }
48 }
```

Question 6 Short answer

A `Scorekeeper` method has this specification: "Moves `amount` points from `giver` to `receiver`." Its body calls `giver.addPoints(-amount)` and `receiver.addPoints(amount)`. Is this method breaking the good-practice rule about changing parameters? Explain.

3.6.4 Returning a Reference

Sometimes a method's answer is an object: "which of these two players is ahead?" In Topic 3.5, a non-void method evaluated its return expression and handed back a copy of the value, so returning an `int` gave the caller a number of its own. What does the caller get when the return type is a class, such as `Player`?

The same thing happens as with parameters. The return expression evaluates to a reference, and that reference is what gets copied back to the caller. The object itself is **not** copied. If the method returns a reference to an object that already exists, the caller's variable becomes another alias of that object.

KEY IDEA

When a method returns an object, it returns a reference to that object, not a new copy of it. The caller and anyone else holding that reference share one object.

WORKED EXAMPLE The leader is not a copy

`Leader` returns one of its two parameters. Watch what `top` refers to after line 8.

LeaderReturn.java

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

```
5     Player ada = new Player("Ada", 12);
6     Player grace = new Player("Grace", 9);
7     Scorekeeper keeper = new Scorekeeper();
8     Player top = keeper.leader(ada, grace);
9     top.addPoints(1);
10    System.out.println(top.getName() + " " + top.getPoints());
11    System.out.println(ada.getName() + " " + ada.getPoints());
12    System.out.println(top == ada);
13  }
14 }
15
16 class Scorekeeper
17 {
18     // Returns whichever player has more points (a if they are tied).
19     public Player leader(Player a, Player b)
20     {
21         if (b.getPoints() > a.getPoints())
22         {
23             return b;
24         }
25         return a;
26     }
27 }
28
29 class Player
30 {
31     private String name;
32     private int points;
33
34     public Player(String playerName, int startPoints)
35     {
36         name = playerName;
37         points = startPoints;
38     }
39
40     public String getName()
41     {
42         return name;
43     }
44
45     public int getPoints()
46     {
47         return points;
48     }
49
50     public void addPoints(int amount)
51     {
52         points = points + amount;
53     }
54 }
```

Lines 5 and 6 build Ada (address A, 12 points) and Grace (address B, 9 points). Line 8 copies address A into `a` and address B into `b`. In `leader`, line 21 asks whether `9 > 12`, which is false, so

line 25 runs `return a;`. The value of the expression `a` is "address A," and that is what comes back and gets stored in `top`:

Variable	Holds	Object at that address
<code>ada</code>	reference to address A	Player "Ada", 12 points
<code>grace</code>	reference to address B	Player "Grace", 9 points
<code>top</code>	reference to address A	(the same object as <code>ada</code>)

Line 9 adds 1 point through `top`, which changes Ada's object to 13 points. Lines 10 and 11 reach that same object through two different variables. Line 12 compares the two references with `==` (Topic 2.6), which is `true` because they hold the same address.

```
Ada 13
Ada 13
true
```

Now vary it: the method returns an **instance variable** instead of a parameter. This `Team` builds its own private `Player` in its constructor, and `getCaptain` returns it.

CaptainLeak.java

```
1 public class CaptainLeak
2 {
3     public static void main(String[] args)
4     {
5         Team robots = new Team("Ada");
6         Player c = robots.getCaptain();
7         c.addPoints(50);
8         System.out.println("Team says: " + robots.getCaptainPoints());
9     }
10 }
11
12 class Team
13 {
14     private Player captain;
15
16     public Team(String captainName)
17     {
18         captain = new Player(captainName, 0);
19     }
20
21     public Player getCaptain()
22     {
23         return captain;
24     }
25
26     public int getCaptainPoints()
27     {
28         return captain.getPoints();
29     }
30 }
31
32 class Player
```

```
33 {
34     private String name;
35     private int points;
36
37     public Player(String playerName, int startPoints)
38     {
39         name = playerName;
40         points = startPoints;
41     }
42
43     public String getName()
44     {
45         return name;
46     }
47
48     public int getPoints()
49     {
50         return points;
51     }
52
53     public void addPoints(int amount)
54     {
55         points = points + amount;
56     }
57 }
```

Team says: 50

Line 23 returns the reference stored in the private instance variable `captain`. Line 6 stores that reference in `c`, so `c` and the team's `captain` are aliases. Line 7 adds 50 points through `c`, and the team's own `captain` object changes, even though `Team` has no method for changing its captain's points. Making `captain` private kept other classes from writing `robots.captain`, but it did not protect the object once a reference to it was handed out.

If the class designer does not want outside code to change the team's captain, `getCaptain` can build and return a new object with the same data:

```
1 return new Player(captain.getName(), captain.getPoints());
```

With that line as the body of `getCaptain`, line 7 changes only the copy, and the program prints `Team says: 0`. This is the return-side partner of the Topic 3.4 advice to store a copy of a mutable object that a constructor receives.

WATCH OUT

Expecting return to hand back a copy. It is tempting to think that a variable like `top` or `c` holds "the answer" and is safe to change. It is not a copy unless the method made one with `new`. `return captain;` and `return a;` hand back an existing object's address, so changes through the returned reference show up everywhere that object is used. If you need an independent object, the method must build one, as in the fix above.

PAUSE AND TRY**Question 7** Tracing

`leader` is the same method as in `LeaderReturn`: it returns `b` only when `b` has more points, and otherwise returns `a`. Write the exact output.

`LeaderTie.java`

```
1 public class LeaderTie
2 {
3     public static void main(String[] args)
4     {
5         Player lee = new Player("Lee", 7);
6         Player mia = new Player("Mia", 7);
7         Scorekeeper keeper = new Scorekeeper();
8         Player top = keeper.leader(lee, mia);
9         top.addPoints(2);
10        Player next = keeper.leader(mia, lee);
11        next.addPoints(1);
12        System.out.println(lee.getPoints() + " " + mia.getPoints());
13        System.out.println(top == next);
14    }
15 }
16
17 class Scorekeeper
18 {
19     // Returns whichever player has more points (a if they are tied).
20     public Player leader(Player a, Player b)
21     {
22         if (b.getPoints() > a.getPoints())
23         {
24             return b;
25         }
26         return a;
27     }
28 }
29
30 class Player
31 {
32     private String name;
33     private int points;
34
35     public Player(String playerName, int startPoints)
36     {
37         name = playerName;
38         points = startPoints;
39     }
40
41     public String getName()
42     {
43         return name;
44     }
45
46     public int getPoints()
```

```
47     {  
48         return points;  
49     }  
50  
51     public void addPoints(int amount)  
52     {  
53         points = points + amount;  
54     }  
55 }
```

Question 8 Multiple choice

In `CaptainLeak`, which replacement for line 23 keeps lines 6 and 7 from changing the team's own captain?

- A. `Player backup = new Player(captain.getName(), captain.getPoints()); return captain;`
- B. `Player result = captain; return result;`
- C. `return new Player(captain.getName(), captain.getPoints());`
- D. `return captain.getPoints();`

3.6.5 Private Data of a Parameter

Suppose `Player` needs a method that answers "is this player ahead of that one?" The method receives another `Player` as a parameter. Inside `Player`, can it read `other.points` directly, or must it call `other.getPoints()`? And what about a `Scorekeeper` method that receives a `Player`?

From Topic 3.3, `private` restricts access to the **declaring class**. The key word is *class*, not *object*. Code written inside the `Player` class may use the private instance variables and private methods of **any** `Player` object it has a reference to, not only the object it was called on. Code in any other class may not, even when it holds a perfectly good reference to a `Player`.

Why draw the line at the class? `private` exists to hide how a class works from code written outside it (encapsulation). Code inside `Player` is already part of that hidden design: whoever writes `Player` knows what `points` means and keeps it correct for every `Player` object. Code in `Scorekeeper` is outside, so it must go through the public methods.

KEY IDEA

A method can use the private data and private methods of an object passed to it only when the parameter's type is the same class the method is written in. Every other class must use public methods such as accessors.

WORKED EXAMPLE Comparing with another player

`isAheadOf` is written inside `Player` and receives another `Player`.

AheadOf.java

```
1 public class AheadOf
2 {
3     public static void main(String[] args)
4     {
5         Player ada = new Player("Ada", 12);
6         Player grace = new Player("Grace", 9);
7         System.out.println(ada.isAheadOf(grace));
8         System.out.println(grace.isAheadOf(ada));
9     }
10 }
11
12 class Player
13 {
14     private String name;
15     private int points;
16
17     public Player(String playerName, int startPoints)
18     {
19         name = playerName;
20         points = startPoints;
21     }
22
23     public boolean isAheadOf(Player other)
24     {
25         return points > other.points;
26     }
27 }
```

On line 7, `ada.isAheadOf(grace)` runs the method for Ada's object, with `other` holding a reference to Grace's object. On line 25, `points` with no dot means the points of the object the method was called on, Ada's 12. `other.points` follows `other` to Grace's object and reads its private `points`, 9. Because the method is in `Player` and `other` is a `Player`, the compiler allows it. `12 > 9` is `true`. Line 8 reverses the roles: `9 > 12` is `false`.

```
true
false
```

The rule covers private methods too. Here `hasScored` is private, and `bothScored`, also inside `Player`, calls it on itself and on the parameter:

BothScored.java

```
1 public class BothScored
2 {
3     public static void main(String[] args)
4     {
5         Player ada = new Player("Ada", 12);
6         Player kai = new Player("Kai", 0);
7         System.out.println(ada.bothScored(kai));
8         System.out.println(ada.bothScored(ada));
9     }
10 }
```

```
11
12 class Player
13 {
14     private String name;
15     private int points;
16
17     public Player(String playerName, int startPoints)
18     {
19         name = playerName;
20         points = startPoints;
21     }
22
23     public boolean bothScored(Player other)
24     {
25         return hasScored() && other.hasScored();
26     }
27
28     private boolean hasScored()
29     {
30         return points > 0;
31     }
32 }
```

```
false
true
```

Line 7: Ada has 12 points but Kai has 0, so `other.hasScored()` is `false` and the result is `false`. Line 8 passes `ada` as the argument to her own method, so both calls check the same object, and the result is `true`.

WATCH OUT

Reaching into private data from a different class. `Scorekeeper` holds a reference to a `Player`, so it seems like it should be able to read the points:

`PrivateFromOutside.java`

```
1 public class PrivateFromOutside
2 {
3     public static void main(String[] args)
4     {
5         Player ada = new Player("Ada", 12);
6         Scorekeeper keeper = new Scorekeeper();
7         System.out.println(keeper.pointsOf(ada));
8     }
9 }
10
11 class Scorekeeper
12 {
13     public int pointsOf(Player p)
14     {
15         return p.points;
16     }
17 }
```

```
17 }
18
19 class Player
20 {
21     private String name;
22     private int points;
23
24     public Player(String playerName, int startPoints)
25     {
26         name = playerName;
27         points = startPoints;
28     }
29
30     public int getPoints()
31     {
32         return points;
33     }
34 }
```

This does not compile. Java reports something like this (the wording can vary between Java versions):

```
PrivateFromOutside.java:15: error: points has private access in Player
```

The method is in `Scorekeeper`, but the parameter's type is `Player`, a different class, so its private variables are off limits. The fix is to use the public accessor: change line 15 to `return p.getPoints();`, and the program prints 12.

PAUSE AND TRY

Question 9 Multiple choice

`SameScoreAccess` does not compile. Which line causes the error?

`SameScoreAccess.java`

```
1 public class SameScoreAccess
2 {
3     public static void main(String[] args)
4     {
5         Player ada = new Player("Ada", 12);
6         Player kai = new Player("Kai", 12);
7         Coach coach = new Coach();
8         System.out.println(ada.sameScore(kai));
9         System.out.println(coach.sameName(ada, kai));
10    }
11 }
12
13 class Coach
14 {
15     public boolean sameName(Player p, Player q)
16     {
```

```

17         return p.name.equals(q.name);
18     }
19 }
20
21 class Player
22 {
23     private String name;
24     private int points;
25
26     public Player(String playerName, int startPoints)
27     {
28         name = playerName;
29         points = startPoints;
30     }
31
32     public boolean sameScore(Player other)
33     {
34         return points == other.points;
35     }
36 }

```

- A. Line 8, because `kai` is passed to a method of a different object
- B. Line 17, because `Coach` uses the private `name` of its `Player` parameters
- C. Line 34, because `sameScore` uses the private `points` of its parameter `other`
- D. Both line 17 and line 34, because private data can be used only for the object a method is called on

Putting it together

This program uses every idea in the chapter. `trade` changes both of its parameters (its specification says to) and returns a reference. `resetToZero` tries to replace its parameter. `isAheadOf` reads a parameter's private data from inside `Player`.

TradeDay.java

```

1  public class TradeDay
2  {
3      public static void main(String[] args)
4      {
5          Player ada = new Player("Ada", 20);
6          Player grace = new Player("Grace", 12);
7          Scorekeeper keeper = new Scorekeeper();
8          Player leader = keeper.trade(ada, grace, 5);
9          leader.addPoints(2);
10         keeper.resetToZero(ada);
11         System.out.println("Ada: " + ada.getPoints());
12         System.out.println("Grace: " + grace.getPoints());
13         System.out.println(leader == grace);
14         System.out.println(grace.isAheadOf(ada));
15     }
16 }

```

```
17
18 class Scorekeeper
19 {
20     // Moves amount points from giver to receiver.
21     // Returns whichever player is ahead afterward (giver if tied).
22     public Player trade(Player giver, Player receiver, int amount)
23     {
24         giver.addPoints(-amount);
25         receiver.addPoints(amount);
26         if (receiver.getPoints() > giver.getPoints())
27         {
28             return receiver;
29         }
30         return giver;
31     }
32
33     // Meant to set p's points to 0 (it does not work).
34     public void resetToZero(Player p)
35     {
36         p = new Player(p.getName(), 0);
37     }
38 }
39
40 class Player
41 {
42     private String name;
43     private int points;
44
45     public Player(String playerName, int startPoints)
46     {
47         name = playerName;
48         points = startPoints;
49     }
50
51     public String getName()
52     {
53         return name;
54     }
55
56     public int getPoints()
57     {
58         return points;
59     }
60
61     public void addPoints(int amount)
62     {
63         points = points + amount;
64     }
65
66     public boolean isAheadOf(Player other)
67     {
68         return points > other.points;
69     }
70 }
```

Ada's object is at address A and Grace's at address B.

Line(s)	What happens	Ada	Grace	Leader
5–6	Two objects are built	20	12	
8, 24–25	<code>giver</code> gets address A, <code>receiver</code> gets address B; mutators change both shared objects	15	17	
26–28	<code>17 > 15</code> is true, so it returns <code>receiver</code> , address B, stored in <code>leader</code>	15	17	B
9	<code>leader</code> is an alias of Grace, so Grace gets 2	15	19	B
10, 36	<code>p</code> gets address A, then is moved to a new 0-point object; the new object is lost when the method ends	15	19	B
11–12	Print both totals	15	19	B
13	<code>leader</code> and <code>grace</code> both hold address B	15	19	B
14	Inside <code>Player</code> , <code>points > other.points</code> is <code>19 > 15</code>	15	19	B

```
Ada: 15
Grace: 19
true
true
```

Notice the two methods that receive `Ada`. `trade` used a mutator through its parameter, and the change stuck. `resetToZero` assigned to its parameter, and nothing changed for the caller. To actually reset a player, `Player` would need a mutator that sets `points` to 0, and `resetToZero` would call it.

Vocabulary

Term	Definition	Example
mutable	Describes an object whose state can be changed after it is created, usually through mutators	A <code>Player</code> with <code>addPoints</code>
specification	A description of what a method or class is supposed to do, which callers rely on	"Returns the total of the two players' points"
side effect	A change a method makes to an object passed to it (or other state) that its specification does not call for	<code>previewBonus</code> adding points to <code>p</code>
reference (review)	The address of an object, which is what a variable of a class type holds	<code>star</code> holds "address A"
alias (review)	One of two or more variables that refer to the same object	<code>p</code> and <code>star</code> during <code>awardBonus(star)</code>

Chapter summary

- When the argument is an object, the parameter gets a copy of the reference. No new object is made, so the parameter and the caller's variable refer to the same object.
- Calling a mutator through the parameter changes the shared object, and the caller sees the change. Assigning a new reference to the parameter changes only the parameter, and the caller sees nothing. A `String` argument can never be changed, because strings are immutable.
- A method or constructor should not change a mutable object it receives unless its specification calls for that change; otherwise it should only read the object with accessors.
- Returning an object returns a reference to it, not a copy. Returning a private instance variable's reference lets outside code change that object; returning a new object built with `new` prevents this.
- Code may use the private variables and private methods of a parameter only when the parameter's type is the class the code is written in.

Review questions

Question 10 Multiple choice

This `Team` constructor stores the `Player` it receives. What does the program print?

`StoredReference.java`

```
1 public class StoredReference
2 {
3     public static void main(String[] args)
4     {
5         Player ada = new Player("Ada", 10);
6         Team robots = new Team(ada);
7         ada.addPoints(5);
8         System.out.println(robots.getCaptainPoints());
9     }
10 }
11
12 class Team
13 {
14     private Player captain;
15
16     public Team(Player firstCaptain)
17     {
18         captain = firstCaptain;
19     }
20
21     public int getCaptainPoints()
22     {
23         return captain.getPoints();
24     }
25 }
26
27 class Player
28 {
29     private String name;
30     private int points;
31
32     public Player(String playerName, int startPoints)
```

```
33     {
34         name = playerName;
35         points = startPoints;
36     }
37
38     public String getName()
39     {
40         return name;
41     }
42
43     public int getPoints()
44     {
45         return points;
46     }
47
48     public void addPoints(int amount)
49     {
50         points = points + amount;
51     }
52 }
```

- A. 10, because the team made its own copy of Ada when it was built
- B. 15, because captain and ada refer to the same object
- C. 5, because only the change after line 6 counts
- D. Nothing; a constructor cannot receive an object as a parameter

Question 11 Tracing

Write the exact output.

PickTrace.java

```
1  public class PickTrace
2  {
3      public static void main(String[] args)
4      {
5          Player ada = new Player("Ada", 5);
6          Player bo = new Player("Bo", 5);
7          Scorekeeper keeper = new Scorekeeper();
8          Player result = keeper.pick(ada, bo);
9          result.addPoints(100);
10         System.out.println(ada.getPoints());
11         System.out.println(bo.getPoints());
12         System.out.println(result == bo);
13     }
14 }
15
16 class Scorekeeper
17 {
18     public Player pick(Player p, Player q)
19     {
20         p.addPoints(1);
21         p = q;
22         p.addPoints(10);
23         return p;
24     }
25 }
```

```
25 }
26
27 class Player
28 {
29     private String name;
30     private int points;
31
32     public Player(String playerName, int startPoints)
33     {
34         name = playerName;
35         points = startPoints;
36     }
37
38     public String getName()
39     {
40         return name;
41     }
42
43     public int getPoints()
44     {
45         return points;
46     }
47
48     public void addPoints(int amount)
49     {
50         points = points + amount;
51     }
52 }
```

Question 12 Debugging

Shop's refund method is supposed to add amount dollars to the customer's wallet, but the program does not compile. Explain the error and correct line 17.

ShopRefund.java

```
1 public class ShopRefund
2 {
3     public static void main(String[] args)
4     {
5         Wallet mine = new Wallet(20);
6         Shop shop = new Shop();
7         shop.refund(mine, 5);
8         System.out.println(mine.getDollars());
9     }
10 }
11
12 class Shop
13 {
14     // Gives amount dollars back to the customer's wallet.
15     public void refund(Wallet w, int amount)
16     {
17         w.dollars = w.dollars + amount;
18     }
19 }
```

```
20
21 class Wallet
22 {
23     private int dollars;
24
25     public Wallet(int startDollars)
26     {
27         dollars = startDollars;
28     }
29
30     public int getDollars()
31     {
32         return dollars;
33     }
34
35     public void addDollars(int amount)
36     {
37         dollars = dollars + amount;
38     }
39 }
```

Question 13 Multiple choice

A student wrote `makeCopy` to give `main` a backup of a player. What does the program print?

FakeCopy.java

```
1 public class FakeCopy
2 {
3     public static void main(String[] args)
4     {
5         Player ada = new Player("Ada", 10);
6         Scorekeeper keeper = new Scorekeeper();
7         Player backup = keeper.makeCopy(ada);
8         backup.addPoints(5);
9         System.out.println(ada.getPoints());
10    }
11 }
12
13 class Scorekeeper
14 {
15     public Player makeCopy(Player p)
16     {
17         Player result = p;
18         return result;
19     }
20 }
21
22 class Player
23 {
24     private String name;
25     private int points;
26
27     public Player(String playerName, int startPoints)
28     {
29         name = playerName;
```

```
30         points = startPoints;
31     }
32
33     public String getName()
34     {
35         return name;
36     }
37
38     public int getPoints()
39     {
40         return points;
41     }
42
43     public void addPoints(int amount)
44     {
45         points = points + amount;
46     }
47 }
```

- A. 10, because backup is a separate copy of Ada
- B. 15, because makeCopy returns the same reference it received
- C. 5, because backup starts with 0 points
- D. Nothing; a method cannot return a parameter

Question 14 Short answer

A classmate says: "When you pass an object to a method, the method gets the actual variable, so it can do anything to the caller's variable, including making it refer to a different object." Using `BonusByReference` and `ReplaceParameter` from this chapter, explain what part of that claim is right and what part is wrong.

Reflection

- ☐ I can explain what a parameter holds when the argument is an object, and why no new object is made. Reread: 3.6.1 Passing an Object Reference
- ☐ I can predict whether a change made inside a method shows up for the caller. Reread: 3.6.2 What a Method Cannot Change
- ☐ I can spot a method or constructor that changes a parameter object its specification does not mention. Reread: 3.6.3 Leave Parameters Unchanged Unless Asked
- ☐ I can trace a method that returns an object and explain why the caller gets the same object, not a copy. Reread: 3.6.4 Returning a Reference
- ☐ I can decide whether a method may use the private data of an object passed to it. Reread: 3.6.5 Private Data of a Parameter

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

16

`grace` refers to one `Player` object that starts with 4 points. Each call copies that reference into `p`, so `p` leads to the same object.

Call	<code>p.getPoints()</code> before line 17	<code>addPoints</code> adds	Points afterward
First (line 7)	4	4	8
Second (line 8)	8	8	16

The second call sees 8, not 4, because the first call already changed the shared object. Line 9 prints 16.

Reread 3.6.1 *Passing an Object Reference* if your output differed.

Question 2

B. A parameter is initialized with a copy of its argument, and the argument `star` holds a reference. So `p` holds a copy of that reference, and both lead to one object.

- **A** is wrong: no `new` runs during the call, so no second object exists. This is the "method gets a photocopy of the object" misconception.
- **C** is wrong: the parameter's type is `Player`, not `int`. The method can get the points by calling `p.getPoints()`, but `p` itself holds a reference.
- **D** is wrong: `p` is a separate variable that holds a copy of the address. Assigning a different object to `p` would change only `p`, as 3.6.2 shows.

Reread 3.6.1 *Passing an Object Reference* if you chose differently.

Question 3

Kai 13

Line	<code>p</code> refers to	Kai's object	"Temp" object
7	Kai's object (address A)	10 points	(none yet)
16	Kai's object	13 points	(none yet)
17	"Temp" object (address B)	13 points	100 points
18	"Temp" object	13 points	103 points

Line 16 runs while `p` still refers to Kai's object, so Kai gets 3 points. Line 17 moves `p` to a new object, so line 18 adds 3 to "Temp", not to Kai. `kai` in `main` still refers to address A, so line 8 prints Kai 13.

Reread 3.6.2 *What a Method Cannot Change* if your output differed.

Question 4

Model answer: The method's parameter holds a copy of the reference to the caller's `String`. A `String` is immutable, so there is no method that changes the text of that object. The only way to get new text is to build a new `String` and assign it to the parameter, but assigning to a parameter changes only the parameter, not the caller's variable. So the caller's variable still refers to the original, unchanged text.

A complete answer mentions both ideas:

- Strings are immutable, so the shared object cannot be changed.
- Assigning a new string to the parameter does not change the caller's variable.

(The method could *return* the new string, and the caller could store it, which is how `String` methods like `substring` work.)

Reread 3.6.2 *What a Method Cannot Change* if your answer differed.

Question 5

The program prints 14 and then 14. The total is right, but line 18 adds Mia's points to Lee's object, so Lee now has 14 points instead of 8. Because `a` holds a copy of `lee`'s reference, the mutator call changes the caller's object, which the specification never asked for.

Corrected body:

```
1 return a.getPoints() + b.getPoints();
```

Now the method only reads both objects through accessors. The program prints 14 and then 8.

Reread 3.6.3 *Leave Parameters Unchanged Unless Asked* if your fix differed.

Question 6

Model answer: No. The rule is to avoid changing a parameter's object *unless the specification requires it*. Here, changing both players' points is the method's whole job, so the changes are exactly what callers expect. They are not side effects.

A complete answer says that the rule depends on the specification, and that this specification asks for both objects to change.

Reread 3.6.3 *Leave Parameters Unchanged Unless Asked* if your answer differed.

Question 7

```
10 7
true
```

Line	What happens	Lee	Mia
8	<code>leader(lee, mia)</code> : is <code>7 > 7</code> ? No, so it returns <code>a</code> , Lee's reference	7	7
9	<code>top</code> refers to Lee, so Lee gets 2	9	7
10	<code>leader(mia, lee)</code> : now <code>a</code> is Mia, <code>b</code> is Lee. Is <code>9 > 7</code> ? Yes, so it returns <code>b</code> , Lee's reference	9	7
11	<code>next</code> refers to Lee, so Lee gets 1	10	7

Both calls returned a reference to Lee's object, so `top == next` is `true`. No call created a new object.

Reread 3.6.4 *Returning a Reference* if your output differed.

Question 8

C. It builds a new `Player` object with the captain's data and returns a reference to that new object. Line 7 then changes the copy, and the team's captain keeps 0 points.

- **A** is wrong: it builds a copy but then returns `captain`, the team's own reference. The copy is never used, so the program still prints `Team says: 50`. Making a copy helps only if the copy is what gets returned.
- **B** is wrong: `result = captain` copies the address, not the object. `result` is an alias, so the same reference comes back. This is the "assignment makes a copy" misconception, and the program still prints `Team says: 50`.
- **D** is wrong: it returns an `int`, which does not match the return type `Player`, so the method does not compile.

Reread 3.6.4 *Returning a Reference* if you chose differently.

Question 9

B. `sameName` is written in `Coach`, but `p` and `q` are `Player` references, a different class, so `p.name` and `q.name` are not allowed. Java reports `name` has private access in `Player`.

- **A** is wrong: passing a reference to any method is always allowed; access rules are about what code may use once it has the reference.
- **C** is wrong: `sameScore` is written inside `Player` and its parameter is a `Player`, so `other.points` is allowed.
- **D** is wrong: `private` is about the class the code is in, not about which object. This choice comes from thinking each object guards its own private data from every other object.

Reread 3.6.5 *Private Data of a Parameter* if you chose differently.

Question 10

B. Line 6 copies `ada`'s reference into `firstCaptain`, and line 18 copies it again into the instance variable `captain`. No new `Player` runs, so there is one object. Line 7 adds 5 points to it through `ada`, and `getCaptainPoints` reads that same object: 15.

- **A** is wrong: making a copy takes `new`. Line 18 copies only the address. This is the advice from Topic 3.4 that the constructor did *not* follow.
- **C** is wrong: `addPoints` adds to the existing 10 points; it does not start the total over.
- **D** is wrong: constructors receive object references exactly the way methods do.

Reread 3.6.1 *Passing an Object Reference* if you chose differently.

Question 11

```
6
115
true
```

Ada's object is address A, Bo's is address B.

Line	p refers to	Ada	Bo	Notes
------	-------------	-----	----	-------

8	A	5	5	p gets A, q gets B
20	A	6	5	Mutator through p changes Ada
21	B	6	5	Only p changes; ada still holds A
22	B	6	15	Mutator through p now changes Bo
23	B	6	15	Returns address B, stored in result
9		6	115	result is an alias of Bo

Line 12 compares two references that both hold address B, so it prints `true`.

Reread 3.6.2 *What a Method Cannot Change* and 3.6.4 *Returning a Reference* if your output differed.

Question 12

Java reports `dollars` has private access in `Wallet` for line 17. `refund` is written in `Shop`, but `w` is a `Wallet`, a different class, so `Shop` code may not use `w.dollars`.

Corrected line 17:

```
1 w.addDollars(amount);
```

`addDollars` is a public mutator of `Wallet`, so `Shop` may call it. Because `w` holds a copy of `mine`'s reference, the change reaches the caller's wallet, and the program prints 25. Changing the wallet here is fine: the specification on line 14 asks for it.

Reread 3.6.5 *Private Data of a Parameter* if your fix differed.

Question 13

B. Line 7 copies `ada`'s reference into `p`. Line 17 copies it into `result`, and line 18 returns it. No `new` ever runs, so `backup`, `result`, `p`, and `ada` all refer to the one object. Line 8 adds 5 to it, and line 9 prints 15.

- **A** is wrong: neither the call, the assignment on line 17, nor `return` copies an object. Only `new` makes one.
- **C** is wrong: no new `Player` is constructed, so nothing starts at 0.
- **D** is wrong: returning a parameter is allowed whenever its type matches the return type, as `leader` did in 3.6.4.

To make a real backup, replace lines 17 and 18 with `return new Player(p.getName(), p.getPoints());`. Then `backup` refers to a new object, and the program prints 10.

Reread 3.6.4 *Returning a Reference* if you chose differently.

Question 14

Model answer: The method does get access to the caller's *object*, because the parameter holds a copy of the same reference. That is why `awardBonus` in `BonusByReference` could add 5 points and `main` saw 15. But the method does not get the caller's *variable*. The parameter is a separate variable holding a copy of the address. In `ReplaceParameter`, `reset` stored a new object in `p`, and `star` in `main` still referred to `Ada` with 10 points. So a method can change the caller's mutable object, but it can never make the caller's variable refer to a different object.

A complete answer includes:

- The parameter holds a copy of the reference, so it reaches the same object, and mutators change it for the caller.
- The parameter is a separate variable, so assigning to it does not change the caller's variable.

Reread *3.6.1 Passing an Object Reference* and *3.6.2 What a Method Cannot Change* if your answer differed.