

3.5 Methods: How to Write Them

Self-Study Guide · About 60 minutes · With answer key

By the end you will be able to

- Read and write a void method inside a class, and explain why a call to it stands alone as a statement.
- Explain what a mutator method does and trace how its parameters change an object's instance variables.
- Read a non-void method's header and return statement, and decide whether the returned value matches the return type.
- Explain what an accessor method does and why the caller receives a copy of the value.
- Trace exactly where a method stops when a return statement runs, including inside an if statement or a loop.
- Explain why changing a primitive parameter inside a method never changes the caller's variable.

What to remember from earlier

- A method header lists a return type, a name, and a parameter list. ([Topic 1.9: Method Signatures, 1.9.5 Method Signatures](#))
- A void method gives nothing back, and a non-void method returns a value that replaces the call. ([Topic 1.9: Method Signatures, 1.9.7 Return Values](#))
- An argument's value is copied into its parameter (call by value). ([Topic 1.9: Method Signatures, 1.9.6 Call by Value](#))
- You call a method on an object with a dot. ([Topic 1.14: Calling Instance Methods, 1.14.2 The Dot Operator](#))
- Private means only code inside the class can use the variable. ([Topic 3.3: Anatomy of a Class, 3.3.3 The public and private Keywords](#))
- A constructor gives each object's instance variables their starting values. ([Topic 3.4: Constructors, 3.4.2 Writing a Constructor](#))

3.5.1 Writing a Void Method in a Class

In Topics 3.3 and 3.4 you built classes with private instance variables and a constructor. A `StepTracker` object can be created holding 4,200 steps, but then it just sits there. Its instance variable is `private`, so `main` cannot even look at it. An object that cannot do anything is not very useful. A class needs **behaviors**: things its objects can do, such as report their steps or record a walk.

You already know how to read methods. In Topic 1.9 you read headers like `public static void printTotal(double ticketPrice, int tickets)`, and in Topic 1.14 you called methods on objects with a dot, as in `word.length()`. Now you write methods yourself, inside your own class, so that your objects can be called the same way.

In plain words: a method written inside a class is a named set of instructions that every object of that class carries. When you call it on one object, the instructions run using **that object's** instance variables.

Here is the simplest kind. A **void method** is a method that does not return a value. Its header has the keyword `void` in front of the method's name, in the spot where a return type would go. Because nothing comes back, a call to a void method is always a statement on its own, like `tracker.printSummary();`.

Here is a void method inside the `StepTracker` class, with each piece of its header labeled:

```
1 public void printSummary()
2 {
3     System.out.println("Steps so far: " + steps);
4 }
```

Piece	What it means
<code>public</code>	Code in other classes, such as <code>main</code> , may call this method (Topic 3.3).
<code>void</code>	The method gives nothing back to the caller.
<code>printSummary</code>	The method's name. Callers use it: <code>tracker.printSummary();</code>
<code>()</code>	The parameter list. It is empty here, so a call passes no arguments.
<code>{ ... }</code>	The method body: the statements that run on each call. They can use the instance variable <code>steps</code> directly by name.

The body uses `steps` even though `steps` is `private`. That is allowed because the method is **inside** the class; `private` only keeps out code from **other** classes. This is the reason classes have methods at all: the method is the class's own, approved way to work with its private data.

Notice also that `printSummary` has no `static`. The methods you read in Topic 1.9 had it, but a method that works with one object's instance variables leaves it out. Topic 3.7 explains what `static` means.

KEY IDEA

A method written inside a class can use that class's instance variables by name. A void method has `void` before its name, returns nothing, and is called as a statement on its own.

WORKED EXAMPLE Calling a void method you wrote

Each example program in this chapter puts two classes in one file so that it runs in one piece. Java allows only one `public` class per file, and its name must match the file name, so the class with `main` is the `public` one and the other class leaves `public` off. In a real project, each class goes in its own file and is declared `public`.

Read the `StepTracker` class first (lines 1–14), then follow `main` (lines 18–24).

FirstVoidMethod.java

```

1  class StepTracker
2  {
3      private int steps;
4
5      public StepTracker(int startSteps)
6      {
7          steps = startSteps;
8      }
9
10     public void printSummary()
11     {
12         System.out.println("Steps so far: " + steps);
13     }
14 }
15
16 public class FirstVoidMethod
17 {
18     public static void main(String[] args)
19     {
20         StepTracker tracker = new StepTracker(4200);
21         System.out.println("Checking tracker...");
22         tracker.printSummary();
23         System.out.println("Done.");
24     }
25 }

```

Step	Line running	What happens	Printed
1	20	The constructor runs (lines 5–8) and sets this object's steps to 4200	
2	21	An ordinary print in <code>main</code>	Checking tracker...
3	22	Call on <code>tracker</code> . Control jumps to line 10, the start of <code>printSummary</code>	
4	12	The body uses <code>tracker</code> 's steps, which is 4200	Steps so far: 4200
5	23	The method's body is finished, so control returns to the line after the call (1.9.3)	Done.

The output is:

```

Checking tracker...
Steps so far: 4200
Done.

```

Line 22 is a statement on its own. Nothing is stored and nothing is printed by `main` on that line; all the printing happens inside the method.

Now vary two things: put an `if` statement inside the method's body, and call the method on two different objects. The body of a method can hold any statements you have learned, including selection and loops from Unit 2.

LibraryCards.java

```
1 class LibraryCard
2 {
3     private int booksOut;
4
5     public LibraryCard(int startBooks)
6     {
7         booksOut = startBooks;
8     }
9
10    public void printStatus()
11    {
12        if (booksOut > 3)
13        {
14            System.out.println(booksOut + " books: limit reached");
15        }
16        else
17        {
18            System.out.println(booksOut + " books: OK");
19        }
20    }
21 }
22
23 public class LibraryCards
24 {
25     public static void main(String[] args)
26     {
27         LibraryCard ana = new LibraryCard(2);
28         LibraryCard ben = new LibraryCard(5);
29         System.out.println("Report");
30         ben.printStatus();
31         ana.printStatus();
32     }
33 }
```

Report

```
5 books: limit reached
2 books: OK
```

`ana` and `ben` are separate objects, each with its own `booksOut` (Topic 3.4). On line 30, `ben.printStatus()` runs the method with **ben's** `booksOut`, which is 5, so the condition `5 > 3` is true and line 14 runs. On line 31, the very same method runs with **ana's** `booksOut`, which is 2, so the `else` branch runs. The method was written once, but what it does depends on which object it was called on. The order of output follows the order of the calls in `main`, not the order the objects were created.

WATCH OUT

Leaving out the return type. Students who know that a void method "returns nothing" sometimes think they can leave the return type out entirely.

MissingReturnType.java

```
1 class StepTracker
2 {
3     private int steps;
4
5     public StepTracker(int startSteps)
6     {
7         steps = startSteps;
8     }
9
10    public printSummary()
11    {
12        System.out.println("Steps so far: " + steps);
13    }
14 }
15
16 public class MissingReturnType
17 {
18     public static void main(String[] args)
19     {
20         StepTracker tracker = new StepTracker(4200);
21         tracker.printSummary();
22     }
23 }
```

Java reports:

```
MissingReturnType.java:10: error: invalid method declaration; return type required
```

Every method header must say what it returns, and "nothing" is said with the word `void`. (Only a constructor has no return type, which is how Java tells a constructor apart from a method, and a constructor's name must match the class.) Fix line 10 by writing `public void printSummary()`.

PAUSE AND TRY**Question 1** Multiple choice

Using the `StepTracker` class from `FirstVoidMethod.java`, a student adds this line to `main`:

```
1 int shown = tracker.printSummary();
```

What happens?

- A. It compiles, prints `Steps so far: 4200`, and stores 4200 in `shown`.
- B. It does not compile, because `printSummary` is a void method, so its call has no value to store.
- C. It compiles and stores 0 in `shown`, because the method returns nothing.
- D. It does not compile, because `steps` is private.

Question 2 Tracing

What does this program print?

TwoTrackers.java

```
1  class StepTracker
2  {
3      private int steps;
4
5      public StepTracker(int startSteps)
6      {
7          steps = startSteps;
8      }
9
10     public void printSummary()
11     {
12         System.out.println("Steps so far: " + steps);
13     }
14 }
15
16 public class TwoTrackers
17 {
18     public static void main(String[] args)
19     {
20         StepTracker maya = new StepTracker(4200);
21         StepTracker leo = new StepTracker(9800);
22         leo.printSummary();
23         maya.printSummary();
24         leo.printSummary();
25     }
26 }
```

3.5.2 Mutator Methods Change an Object

A step tracker that only reports 4,200 steps forever is broken. When you walk, the count has to go up. Could `main` just add to the variable directly? Try it:

PrivateAccess.java

```
1 class StepTracker
2 {
3     private int steps;
4
5     public StepTracker(int startSteps)
6     {
7         steps = startSteps;
8     }
9 }
10
11 public class PrivateAccess
12 {
13     public static void main(String[] args)
14     {
15         StepTracker tracker = new StepTracker(4200);
16         tracker.steps = tracker.steps + 500;
17     }
18 }
```

Java refuses: `PrivateAccess.java:16: error: steps has private access in StepTracker`. As Topic 3.3 explained, `private` keeps code in other classes from touching the instance variable. That protection is on purpose, but it means the class itself must provide a way to change the value.

In plain words: the class offers a public "change it" button, and anyone who wants the value changed has to press that button. The class decides exactly what pressing it does.

A **mutator method** (also called a **modifier method**) is a method that changes the values of an object's instance variables. A mutator is often a void method, because its job is to change the object, not to hand back an answer.

Most mutators need to know **how much** to change. Recall from Topic 1.9 that a **parameter** is a variable in the header that receives the argument's value when the method is called. A method receives values through its parameters and uses those values to do its job:

```
1 public void addSteps(int newSteps)
2 {
3     steps = steps + newSteps;
4 }
```

The parameter `newSteps` holds whatever number the caller passes. The body adds it to the instance variable `steps` and stores the result back in `steps`. Because `steps` belongs to the object, the new value is still there after the method ends. Notice the parameter has a different name from the instance variable; that keeps it clear which one each line means.

KEY IDEA

A mutator method changes an object's instance variables, and it is often void. Its parameters bring in the values it needs; the changes it makes to instance variables last after the method ends.

WORKED EXAMPLE Watching a mutator change the object

Follow `tracker`'s `steps` through each call.

`AddSteps.java`

```

1  class StepTracker
2  {
3      private int steps;
4
5      public StepTracker(int startSteps)
6      {
7          steps = startSteps;
8      }
9
10     public void addSteps(int newSteps)
11     {
12         steps = steps + newSteps;
13     }
14
15     public void printSummary()
16     {
17         System.out.println("Steps so far: " + steps);
18     }
19 }
20
21 public class AddSteps
22 {
23     public static void main(String[] args)
24     {
25         StepTracker tracker = new StepTracker(4200);
26         tracker.addSteps(2500);
27         tracker.printSummary();
28         tracker.addSteps(800);
29         tracker.printSummary();
30     }
31 }

```

Line running	What happens	newSteps	tracker's steps	Printed
25	The constructor sets <code>steps</code> to 4200		4200	
26 → 12	<code>addSteps(2500)</code> : <code>newSteps</code> receives 2500; 4200 + 2500 is 6700	2500	6700	
27 → 17	<code>printSummary</code> reads the updated value		6700	Steps so far: 6700
28 → 12	<code>addSteps(800)</code> : <code>newSteps</code> receives 800; 6700 + 800 is 7500	800	7500	
29 → 17	<code>printSummary</code> again		7500	Steps so far: 7500

The `newSteps` column is blank between calls because a parameter exists only while its method is running. Each call gets a fresh `newSteps` holding that call's argument. The instance variable, by contrast, keeps its value from one call to the next: the second call starts from 6700, not 4200.

Now vary something: a mutator does not need a parameter if it always makes the same change. Here a `reset` method sets the count back to 0, and the order of the calls matters.

ResetSteps.java

```
1 class StepTracker
2 {
3     private int steps;
4
5     public StepTracker(int startSteps)
6     {
7         steps = startSteps;
8     }
9
10    public void addSteps(int newSteps)
11    {
12        steps = steps + newSteps;
13    }
14
15    public void reset()
16    {
17        steps = 0;
18    }
19
20    public void printSummary()
21    {
22        System.out.println("Steps so far: " + steps);
23    }
24 }
25
26 public class ResetSteps
27 {
28     public static void main(String[] args)
29     {
30         StepTracker tracker = new StepTracker(4200);
31         tracker.addSteps(1000);
32         tracker.reset();
33         tracker.addSteps(300);
34         tracker.printSummary();
35     }
36 }
```

Steps so far: 300

Line 31 raises `steps` from 4200 to 5200. Line 32 calls `reset`, which replaces 5200 with 0 no matter what it was. Line 33 adds 300 to 0. If lines 32 and 33 were swapped, the 300 would be wiped out and the program would print `Steps so far: 0`. A mutator changes the object's state at the moment it is called, so later calls start from whatever state earlier calls left behind.

WATCH OUT

Storing the answer in the parameter instead of the instance variable. This version of `addSteps` computes the right sum but puts it in the wrong place:

`MutatorWrong.java`

```
1 class StepTracker
2 {
3     private int steps;
4
5     public StepTracker(int startSteps)
6     {
7         steps = startSteps;
8     }
9
10    public void addSteps(int newSteps)
11    {
12        newSteps = steps + newSteps;
13    }
14
15    public void printSummary()
16    {
17        System.out.println("Steps so far: " + steps);
18    }
19 }
20
21 public class MutatorWrong
22 {
23     public static void main(String[] args)
24     {
25         StepTracker tracker = new StepTracker(4200);
26         tracker.addSteps(2500);
27         tracker.printSummary();
28     }
29 }
```

It compiles and prints `Steps so far: 4200`, as if the walk never happened. Line 12 stores 6700 in `newSteps`, the parameter. The parameter disappears when the method ends, and `steps` was never assigned, so the object is unchanged. Students make this mistake when they think of "the variable in the method" as the one that matters. The variable on the **left** of `=` is the one that changes, so a mutator must assign to the instance variable: `steps = steps + newSteps`; as in `AddSteps.java`.

PAUSE AND TRY**Question 3** Tracing

What does this program print? Keep a separate column for each object.

`ThermostatTrace.java`

```
1 class Thermostat
2 {
```

```
3     private int temperature;
4
5     public Thermostat(int startTemperature)
6     {
7         temperature = startTemperature;
8     }
9
10    public void raise(int degrees)
11    {
12        temperature = temperature + degrees;
13    }
14
15    public void lower(int degrees)
16    {
17        temperature = temperature - degrees;
18    }
19
20    public void printSetting()
21    {
22        System.out.println("Set to " + temperature);
23    }
24 }
25
26 public class ThermostatTrace
27 {
28     public static void main(String[] args)
29     {
30         Thermostat hall = new Thermostat(68);
31         Thermostat office = new Thermostat(70);
32         hall.raise(4);
33         office.lower(3);
34         hall.lower(1);
35         hall.printSetting();
36         office.printSetting();
37     }
38 }
```

Question 4 Debugging

A tip jar should add each tip to its running total. Starting at 10.0, after tips of 2.5 and 4.0 it should print `Tips: 16.5`, but it prints `Tips: 4.0`. Find the error and correct it.

TipJarBug.java

```
1 class TipJar
2 {
3     private double total;
4
5     public TipJar(double startTotal)
6     {
7         total = startTotal;
8     }
9
10    public void addTip(double tip)
11    {
12        total = tip;
13    }
14
15    public void printTotal()
16    {
17        System.out.println("Tips: " + total);
18    }
19 }
20
21 public class TipJarBug
22 {
23     public static void main(String[] args)
24     {
25         TipJar jar = new TipJar(10.0);
26         jar.addTip(2.5);
27         jar.addTip(4.0);
28         jar.printTotal();
29     }
30 }
```

3.5.3 Non-Void Methods Return a Value

Suppose a fitness app needs to know how many steps are left until the daily goal, so it can show a progress bar or work out how many laps remain. A void method could print the number, but printed text goes to the screen, not back into the program. `main` could not do math with it. The class needs a method that **hands back** a number.

In Topic 1.9 you used methods like `Math.sqrt` that return a value, and you saw that Java uses the returned value in place of the call. Now you write the method that does the returning.

A **non-void method** is a method that returns exactly one value to its caller. Its header puts a **return type**, such as `int`, `double`, or `boolean`, in the spot where a void method has `void`. Its body must contain a **return statement**: the keyword `return` followed by an expression.

```
1 public int stepsToGoal()
2 {
3     return goal - steps;
4 }
```

When `return goal - steps;` runs, Java first **evaluates** the expression to a single value, then sends that value back to the caller. What travels back is the value itself, a number like 5800, not the variables `goal` or `steps`. This is called **return by value**.

The value must be **compatible** with the return type, meaning it must be a value Java would let you assign to a variable of that type. The rules are the ones from Topics 1.2 and 1.5: an `int` value is fine in a method that returns `double` (Java widens it automatically), but a `double` value in a method that returns `int` would lose its decimal part, so Java refuses to compile it.

KEY IDEA

A non-void method's header names a return type instead of `void`. Its return statement evaluates an expression and sends that one value back to the caller, and the value must be compatible with the return type.

WORKED EXAMPLE Writing and using an int method

`stepsToGoal` returns the steps still needed. Watch the two different ways `main` uses the returned value.

StepsToGoal.java

```
1 class StepTracker
2 {
3     private int steps;
4     private int goal;
5
6     public StepTracker(int startSteps, int dailyGoal)
7     {
8         steps = startSteps;
9         goal = dailyGoal;
10    }
11
12    public int stepsToGoal()
13    {
14        return goal - steps;
15    }
16 }
17
18 public class StepsToGoal
19 {
20     public static void main(String[] args)
21     {
22         StepTracker tracker = new StepTracker(4200, 10000);
23         int left = tracker.stepsToGoal();
24         System.out.println("Steps left: " + left);
25         System.out.println("Laps left: " + tracker.stepsToGoal() / 500);
26     }
27 }
```

```
Steps left: 5800
Laps left: 11
```

1. Line 22 creates a tracker with `steps` 4200 and `goal` 10000.

2. Line 23 calls `stepsToGoal`. Control jumps to line 14, which evaluates `10000 - 4200` to 5800 and returns it. Back on line 23, the call is replaced by 5800, so the statement becomes `int left = 5800;`. The value is **stored**.
3. Line 24 prints `Steps left: 5800`.
4. Line 25 calls the method again. It returns 5800 again (nothing has changed the object), and the call is replaced by that value: `"Laps left: " + 5800 / 500`. Integer division gives 11, so it prints `Laps left: 11`. This time the value is **used in an expression** and never stored.

The return type `int` matches: `goal - steps` is an `int` minus an `int`, which is an `int`.

Now vary the return type. The same class can have methods that return a `double` and a `boolean`.

GoalChecks.java

```
1 class StepTracker
2 {
3     private int steps;
4     private int goal;
5
6     public StepTracker(int startSteps, int dailyGoal)
7     {
8         steps = startSteps;
9         goal = dailyGoal;
10    }
11
12    public double milesWalked()
13    {
14        return steps / 2000.0;
15    }
16
17    public boolean reachedGoal()
18    {
19        return steps >= goal;
20    }
21 }
22
23 public class GoalChecks
24 {
25     public static void main(String[] args)
26     {
27         StepTracker tracker = new StepTracker(4200, 10000);
28         double miles = tracker.milesWalked();
29         System.out.println(miles);
30         if (tracker.reachedGoal())
31         {
32             System.out.println("Goal met!");
33         }
34         else
35         {
36             System.out.println("Keep walking.");
37         }
38     }
39 }
```

2.1

Keep walking.

- `milesWalked` (lines 12–15) returns `steps / 2000.0`. Dividing by `2000.0` makes it floating-point division (Topic 1.3), so `4200 / 2000.0` is `2.1`, a `double`, which matches the return type `double`.
- `reachedGoal` (lines 17–20) returns `steps >= goal`. A relational expression produces a `boolean` (Topic 2.2), so the method returns `false` here, since `4200` is not at least `10000`. No `if` is needed inside the method: the comparison's result **is** the answer.
- Line 30 uses the returned `boolean` directly as an `if` condition. Since it is `false`, the `else` branch prints `Keep walking`.

Finally, the widening rule in action. Here `length * width` is an `int`, but the method returns `double`:

GardenArea.java

```
1 class Garden
2 {
3     private int length;
4     private int width;
5
6     public Garden(int gardenLength, int gardenWidth)
7     {
8         length = gardenLength;
9         width = gardenWidth;
10    }
11
12    public double area()
13    {
14        return length * width;
15    }
16 }
17
18 public class GardenArea
19 {
20     public static void main(String[] args)
21     {
22         Garden plot = new Garden(3, 4);
23         System.out.println(plot.area());
24     }
25 }
```

It prints `12.0`. Java computes $3 \times 4 = 12$ as an `int`, then widens it to `12.0` to match the return type, just as `double area = 12; would`.

WATCH OUT

Returning a value that does not fit the return type. Here the header promises an `int`, but the expression is a `double`:

LossyReturn.java

```
1 class StepTracker
2 {
3     private int steps;
4
5     public StepTracker(int startSteps)
6     {
7         steps = startSteps;
8     }
9
10    public int milesWalked()
11    {
12        return steps / 2000.0;
13    }
14 }
15
16 public class LossyReturn
17 {
18     public static void main(String[] args)
19     {
20         StepTracker tracker = new StepTracker(4200);
21         System.out.println(tracker.milesWalked());
22     }
23 }
```

Java reports:

```
LossyReturn.java:12: error: incompatible types: possible lossy conversion from
double to int
```

`steps / 2000.0` is 2.1, and an `int` cannot hold the `.1`. Java will not silently throw away part of a value. Students make this mistake by picking the return type before thinking about what the expression produces. Fix it by making the return type match the value:

`public double milesWalked()`. (A cast, `(int) (steps / 2000.0)`, would also compile, but only if dropping the decimal part is truly what you want.)

PAUSE AND TRY**Question 5** Multiple choice

A `StepTracker` has `int` instance variables `steps` and `goal`. Which return statement compiles inside `public boolean overHalfway()`?

- A. `return goal - steps;`
- B. `return steps > goal / 2;`
- C. `return steps / 2.0;`
- D. `return "yes";`

Question 6 Tracing

What does this program print?

PizzaTrace.java

```
1  class PizzaOrder
2  {
3      private int pizzas;
4
5      public PizzaOrder(int pizzaCount)
6      {
7          pizzas = pizzaCount;
8      }
9
10     public int slices()
11     {
12         return pizzas * 8;
13     }
14
15     public double totalCost()
16     {
17         return pizzas * 12.5;
18     }
19 }
20
21 public class PizzaTrace
22 {
23     public static void main(String[] args)
24     {
25         PizzaOrder order = new PizzaOrder(3);
26         int perPerson = order.slices() / 5;
27         System.out.println(perPerson);
28         System.out.println(order.totalCost() + 4);
29     }
30 }
```

3.5.4 Accessor Methods Share a Copy

Private instance variables protect an object's data from careless changes, but sometimes other code legitimately needs to **know** a value. A leaderboard needs each tracker's step count; a report needs the goal. `main` cannot read `tracker.steps` directly (that is the same `private` access error from 3.5.2). How can other classes look at a value without being able to change it?

In plain words: the class answers questions about itself. You ask, "What are your steps?" and it tells you the number. Telling you the number does not give you the power to change it, the same way a friend telling you their locker combination does not move the dial.

An **accessor method** is a method that lets code in other classes obtain a **copy** of an instance variable's value. An accessor is always a non-void method, because its whole job is to return something. By convention its name is `get` followed by the variable's name, so accessors are often called "getters":

```

1 public int getSteps()
2 {
3     return steps;
4 }

```

Because of return by value (3.5.3), `return steps;` sends back the number stored in `steps`, not the variable itself. The caller gets its own copy of that number. (Accessors can also report class variables, which belong to a whole class instead of one object; those arrive in Topic 3.7.)

Accessors and mutators are easy to mix up, so compare them side by side:

Question	Accessor	Mutator
Job	Report an instance variable's value	Change instance variables
Return type	Always non-void (<code>int getSteps()</code>)	Often void (<code>void addSteps(int newSteps)</code>)
Changes the object?	No	Yes
Typical call	<code>int s = tracker.getSteps();</code>	<code>tracker.addSteps(500);</code>

KEY IDEA

An accessor method is a non-void method that returns a copy of an instance variable's value, so other classes can read it without being able to change it.

WORKED EXAMPLE Reading private data through accessors

This `StepTracker` has two accessors and one mutator. Watch how the accessor's answer changes after the mutator runs.

`Accessors.java`

```

1 class StepTracker
2 {
3     private int steps;
4     private int goal;
5
6     public StepTracker(int startSteps, int dailyGoal)
7     {
8         steps = startSteps;
9         goal = dailyGoal;
10    }
11
12    public int getSteps()
13    {
14        return steps;
15    }
16
17    public int getGoal()
18    {
19        return goal;
20    }
21
22    public void addSteps(int newSteps)

```

```
23     {
24         steps = steps + newSteps;
25     }
26 }
27
28 public class Accessors
29 {
30     public static void main(String[] args)
31     {
32         StepTracker tracker = new StepTracker(4200, 10000);
33         System.out.println(tracker.getSteps());
34         tracker.addSteps(3000);
35         System.out.println(tracker.getSteps());
36         System.out.println(tracker.getGoal() - tracker.getSteps());
37     }
38 }
```

```
4200
7200
2800
```

Line	What happens	Printed
33	getSteps returns a copy of steps, which is 4200	4200
34	The mutator adds 3000; steps is now 7200	
35	getSteps returns a copy of the current value, 7200	7200
36	Both accessors are called: getGoal() returns 10000, getSteps() returns 7200, then 10000 - 7200	2800

An accessor always reports the value at the moment it is called. It does not remember an old answer. Line 36 also shows that `main` can do its own math with the returned copies.

Now vary it: what happens if the caller changes the copy it received?

AccessorCopy.java

```
1 class StepTracker
2 {
3     private int steps;
4
5     public StepTracker(int startSteps)
6     {
7         steps = startSteps;
8     }
9
10    public int getSteps()
11    {
12        return steps;
13    }
14 }
15
16 public class AccessorCopy
17 {
18     public static void main(String[] args)
19     {
20         StepTracker tracker = new StepTracker(4200);
21         int mySteps = tracker.getSteps();
22         mySteps = 0;
23         System.out.println(mySteps);
24         System.out.println(tracker.getSteps());
25     }
26 }
```

```
0
4200
```

Line 21 stores the returned copy, 4200, in a new variable `mySteps` that belongs to `main`. Line 22 sets `mySteps` to 0. That changes only `main`'s variable. The object's `steps` is a different variable in a different place, and it still holds 4200, which line 24 confirms. This is exactly why accessors are safe: they share a value, not access.

Variable	After line 21	After line 22
<code>mySteps</code> (in <code>main</code>)	4200	0
<code>tracker</code> 's <code>steps</code>	4200	4200

WATCH OUT

Trying to change a value through an accessor call. A student wants to add 500 steps and writes:

AccessorAssign.java

```
1 class StepTracker
2 {
3     private int steps;
4
5     public StepTracker(int startSteps)
6     {
7         steps = startSteps;
8     }
9
10    public int getSteps()
11    {
12        return steps;
13    }
14 }
15
16 public class AccessorAssign
17 {
18     public static void main(String[] args)
19     {
20         StepTracker tracker = new StepTracker(4200);
21         tracker.getSteps() += 500;
22     }
23 }
```

Java reports:

```
AccessorAssign.java:21: error: unexpected type
    required: variable
    found:     value
```

The message says it exactly: `+=` needs a **variable** on its left, but `tracker.getSteps()` is replaced by a **value**, the number 4200. Writing `4200 += 500;` makes no sense, and neither does this. An accessor only reads. To change the object, call a mutator: `tracker.addSteps(500);`.

PAUSE AND TRY

Question 7 Multiple choice

A PiggyBank class has the instance variable `private double balance;`. Which method is a correct accessor for `balance`?

- A. `public double getBalance() { return balance; }`
- B. `public void getBalance() { System.out.println(balance); }`
- C. `public double getBalance(double amount) { balance = amount; return balance; }`
- D. `public int getBalance() { return balance; }`

Question 8 Short answer

Predict what this program prints, and explain why the second line is not 7.0.

PiggyCopy.java

```
1 class PiggyBank
2 {
3     private double balance;
4
5     public PiggyBank(double startBalance)
6     {
7         balance = startBalance;
8     }
9
10    public double getBalance()
11    {
12        return balance;
13    }
14 }
15
16 public class PiggyCopy
17 {
18     public static void main(String[] args)
19     {
20         PiggyBank bank = new PiggyBank(12.0);
21         double money = bank.getBalance();
22         money = money - 5.0;
23         System.out.println(money);
24         System.out.println(bank.getBalance());
25     }
26 }
```

3.5.5 A Return Statement Ends the Method

A tracker awards a badge: level 3 for at least 10,000 steps, level 2 for at least 5,000, and level 1 otherwise. As soon as the method knows the answer, it should stop and report it. Checking the other cases would be wasted work, and might even pick the wrong level.

In Topic 1.9 you saw that when a method finishes, control returns to the point of the call. The `return` keyword is how a method finishes **on purpose**, at any point you choose.

The `return` keyword sends the flow of control back to the place where the method was called, **immediately**. Two consequences follow:

- **Nothing after it runs.** Any statement that comes after a `return` statement in the same run of the method is never executed.
- **It exits an `if` or a loop too.** If a `return` statement runs inside an `if` statement or a loop, it does not just leave that statement; it ends the whole method on the spot. A loop does not finish its remaining iterations.

A void method can use `return` as well, written with no value: `return;`. It simply means "stop here." (The same is true inside a constructor, though it is rarely needed there.)

KEY IDEA

When a return statement runs, the method ends immediately and control goes back to the caller. This holds even inside an if statement or a loop: the rest of the method, including any remaining loop iterations, is skipped.

WORKED EXAMPLE Returning from inside if statements

`badgeLevel` has three return statements. For each call, find the **first** one that runs.

BadgeLevel.java

```

1  class StepTracker
2  {
3      private int steps;
4
5      public StepTracker(int startSteps)
6      {
7          steps = startSteps;
8      }
9
10     public int badgeLevel()
11     {
12         if (steps >= 10000)
13         {
14             return 3;
15         }
16         if (steps >= 5000)
17         {
18             return 2;
19         }
20         return 1;
21     }
22 }
23
24 public class BadgeLevel
25 {
26     public static void main(String[] args)
27     {
28         StepTracker sam = new StepTracker(7000);
29         StepTracker kim = new StepTracker(12000);
30         StepTracker raj = new StepTracker(2500);
31         System.out.println(sam.badgeLevel());
32         System.out.println(kim.badgeLevel());
33         System.out.println(raj.badgeLevel());
34     }
35 }

```

2
3
1

Call (line)	steps	steps >= 10000?	steps >= 5000?	Return that runs
-------------	-------	-----------------	----------------	------------------

sam (31)	7000	false: skip line 14	true	line 18: returns 2
kim (32)	12000	true	never checked	line 14: returns 3
raj (33)	2500	false	false	line 20: returns 1

Look at kim. 12000 is also at least 5000, so if the method kept going it would reach `return 2;`. It does not: `return 3;` ends the method at line 14, so line 16 is never even checked. Look at sam: line 20, `return 1;`, sits after the `if` statements with no condition, yet it does not run, because `return 2;` already ended the method. Line 20 runs only when neither `if` returned first. That is why this method needs no `else`.

Now vary it: put the `return` inside a loop. `extraDaysNeeded` checks day by day whether walking `stepsPerDay` more each day for that many days reaches the goal, for up to a week.

DaysToGoal.java

```

1  class StepTracker
2  {
3      private int steps;
4      private int goal;
5
6      public StepTracker(int startSteps, int dailyGoal)
7      {
8          steps = startSteps;
9          goal = dailyGoal;
10     }
11
12     public int extraDaysNeeded(int stepsPerDay)
13     {
14         for (int day = 1; day <= 7; day++)
15         {
16             if (steps + day * stepsPerDay >= goal)
17             {
18                 return day;
19             }
20         }
21         return -1;
22     }
23 }
24
25 public class DaysToGoal
26 {
27     public static void main(String[] args)
28     {
29         StepTracker tracker = new StepTracker(4200, 10000);
30         System.out.println(tracker.extraDaysNeeded(2000));
31         System.out.println(tracker.extraDaysNeeded(500));
32     }
33 }

```

```

3
-1

```

Call on line 30, `stepsPerDay` is 2000, `steps` is 4200, `goal` is 10000:

day	$4200 + \text{day} * 2000$	At least 10000?	What happens
1	6200	false	loop continues
2	8200	false	loop continues
3	10200	true	line 18 returns 3

The loop was set up to run up to 7 times, but the `return` on line 18 stops it after 3 and exits the method. Days 4 to 7 are never checked, and `return -1;` on line 21 never runs.

Call on line 31, `stepsPerDay` is 500: even on day 7, $4200 + 3500 = 7700$ is short of the goal, so the `if` is false every time. The loop ends normally after `day` reaches 8, and control falls through to line 21, which returns `-1`. Using a value that could never be a real answer (no one needs `-1` days) is a common way to report "not found".

A void method can stop early with `return;`. Here `addSteps` refuses a negative number of steps:

EarlyVoidReturn.java

```

1  class StepTracker
2  {
3      private int steps;
4
5      public StepTracker(int startSteps)
6      {
7          steps = startSteps;
8      }
9
10     public void addSteps(int newSteps)
11     {
12         if (newSteps < 0)
13         {
14             System.out.println("Ignored: " + newSteps);
15             return;
16         }
17         steps = steps + newSteps;
18         System.out.println("Added: " + newSteps);
19     }
20
21     public int getSteps()
22     {
23         return steps;
24     }
25 }
26
27 public class EarlyVoidReturn
28 {
29     public static void main(String[] args)
30     {
31         StepTracker tracker = new StepTracker(4200);
32         tracker.addSteps(-300);
33         tracker.addSteps(600);
34         System.out.println(tracker.getSteps());
35     }
36 }

```

```
Ignored: -300
Added: 600
4800
```

On line 32, `newSteps` is `-300`, so the `if` is true: line 14 prints, and `return`; on line 15 ends the method. Lines 17 and 18 are skipped, so `steps` stays 4200. On line 33, `newSteps` is 600, the `if` is false, and lines 17 and 18 run, making `steps` 4800.

WATCH OUT

Writing code after a return statement. Code placed directly after a `return` in the same block can never run, and Java treats that as an error:

Unreachable.java

```
1 class StepTracker
2 {
3     private int steps;
4     private int goal;
5
6     public StepTracker(int startSteps, int dailyGoal)
7     {
8         steps = startSteps;
9         goal = dailyGoal;
10    }
11
12    public int stepsToGoal()
13    {
14        return goal - steps;
15        System.out.println("Calculated steps left");
16    }
17 }
18
19 public class Unreachable
20 {
21     public static void main(String[] args)
22     {
23         StepTracker tracker = new StepTracker(4200, 10000);
24         System.out.println(tracker.stepsToGoal());
25     }
26 }
```

```
Unreachable.java:15: error: unreachable statement
```

The message would never print, so Java stops you. Students write this when they think of `return` as "remember the answer" instead of "leave now." Fix it by moving the print **above** the `return`.

Leaving a path with no return. A non-void method must return a value on every path through it. Take away the last line of `badgeLevel`:

MissingReturn.java

```
1 class StepTracker
2 {
3     private int steps;
4
5     public StepTracker(int startSteps)
6     {
7         steps = startSteps;
8     }
9
10    public int badgeLevel()
11    {
12        if (steps >= 10000)
13        {
14            return 3;
15        }
16        if (steps >= 5000)
17        {
18            return 2;
19        }
20    }
21 }
22
23 public class MissingReturn
24 {
25     public static void main(String[] args)
26     {
27         StepTracker tracker = new StepTracker(7000);
28         System.out.println(tracker.badgeLevel());
29     }
30 }
```

MissingReturn.java:20: error: missing return statement

Java points at the closing brace on line 20. If `steps` were 2500, both conditions would be false and control would reach the end of the method with no value to send back. Java checks this before the program runs, even though this particular object has 7000 steps. Fix it by adding `return 1;` after the second `if`, so every path returns.

PAUSE AND TRY

Question 9 Tracing

What does this program print? Watch where each call's loop stops.

SavingsTrace.java

```
1 class SavingsJar
2 {
3     private int dollars;
4
5     public SavingsJar(int startDollars)
```

```
6      {
7          dollars = startDollars;
8      }
9
10     public int weeksUntil(int target, int weekly)
11     {
12         int total = dollars;
13         int week = 0;
14         while (week < 10)
15         {
16             if (total >= target)
17             {
18                 return week;
19             }
20             week++;
21             total = total + weekly;
22         }
23         return 10;
24     }
25 }
26
27 public class SavingsTrace
28 {
29     public static void main(String[] args)
30     {
31         SavingsJar jar = new SavingsJar(15);
32         System.out.println(jar.weeksUntil(40, 10));
33         System.out.println(jar.weeksUntil(10, 5));
34     }
35 }
```

Question 10 Multiple choice

What does this program print?

ScoreCheck.java

```
1 class Quiz
2 {
3     private int score;
4
5     public Quiz(int startScore)
6     {
7         score = startScore;
8     }
9
10    public void addBonus(int bonus)
11    {
12        if (score + bonus > 100)
13        {
14            score = 100;
15            return;
16        }
17        score = score + bonus;
```

```
18         System.out.println("Bonus added");
19     }
20
21     public int getScore()
22     {
23         return score;
24     }
25 }
26
27 public class ScoreCheck
28 {
29     public static void main(String[] args)
30     {
31         Quiz quiz = new Quiz(95);
32         quiz.addBonus(10);
33         System.out.println(quiz.getScore());
34     }
35 }
```

- A. Bonus added then 100
- B. 100
- C. Bonus added then 105
- D. 105

3.5.6 Parameters Receive Copies

When you hand a method a variable, can the method change that variable? In 3.5.2 a mutator changed the object's instance variable, and the change lasted. A parameter is also a variable the method can assign to. So if a method changes its parameter, does the caller's variable change too?

Topic 1.9 answered this for the methods you read there: Java uses **call by value**. The same rule holds for every method you write in a class.

In plain words: when you call a method, Java photocopies the argument's value and gives the copy to the parameter. The method can scribble on its copy all it likes; your original is untouched.

When an argument is a primitive value, such as an `int`, `double`, or `boolean`, the parameter is **initialized with a copy** of that value. The parameter and the caller's variable are two separate variables that happen to start with the same value. Changing the parameter has no effect on the argument.

This is different from changing an instance variable. An instance variable belongs to the object and outlives the call. A parameter belongs to one run of the method and disappears when the method ends.

KEY IDEA

A primitive argument's value is copied into the parameter. The method may change its parameter, but the caller's variable keeps its value; only changes to instance variables last after the method ends.

WORKED EXAMPLE A deposit that tries to empty the caller's wallet

`deposit` adds `amount` to the balance and then sets `amount` to 0. Does `main`'s `cash` become 0?

`DepositCopy.java`

```

1  class PiggyBank
2  {
3      private double balance;
4
5      public PiggyBank(double startBalance)
6      {
7          balance = startBalance;
8      }
9
10     public void deposit(double amount)
11     {
12         balance = balance + amount;
13         amount = 0;
14     }
15
16     public double getBalance()
17     {
18         return balance;
19     }
20 }
21
22 public class DepositCopy
23 {
24     public static void main(String[] args)
25     {
26         PiggyBank bank = new PiggyBank(5.0);
27         double cash = 20.0;
28         bank.deposit(cash);
29         System.out.println(cash);
30         System.out.println(bank.getBalance());
31     }
32 }

```

20.0
25.0

Picture the variables at each moment:

Moment	cash (main)	amount (parameter)	bank's balance
After line 27	20.0	(does not exist)	5.0
Call on line 28 begins	20.0	20.0 (a copy)	5.0

After line 12	20.0	20.0	25.0
After line 13	20.0	0.0	25.0
Back on line 29	20.0	(gone)	25.0

Line 12 changes an instance variable, so that change lasts. Line 13 changes only the parameter, the copy, and the copy disappears when the method ends. `cash` was never touched, so line 29 prints `20.0`.

Now vary it: let a loop count the parameter down to 0. `walkLaps` adds 400 steps per lap.

WalkLaps.java

```

1  class StepTracker
2  {
3      private int steps;
4
5      public StepTracker(int startSteps)
6      {
7          steps = startSteps;
8      }
9
10     public void walkLaps(int laps)
11     {
12         while (laps > 0)
13         {
14             steps = steps + 400;
15             laps--;
16         }
17     }
18
19     public int getSteps()
20     {
21         return steps;
22     }
23 }
24
25 public class WalkLaps
26 {
27     public static void main(String[] args)
28     {
29         StepTracker tracker = new StepTracker(0);
30         int plannedLaps = 3;
31         tracker.walkLaps(plannedLaps);
32         System.out.println(plannedLaps);
33         System.out.println(tracker.getSteps());
34     }
35 }

```

```

3
1200

```

Inside the method, `laps` starts as a copy of 3. The loop runs while `laps > 0`, adding 400 to `steps` and decreasing `laps` each time: `3 → 2 → 1 → 0`. After three iterations `laps` is 0, the loop ends, and

steps is 1200. Back in `main`, `plannedLaps` still holds 3, because the loop counted down the copy, not the original. Using a parameter as a counter is perfectly safe for exactly this reason.

WATCH OUT

Trying to send an answer back through a parameter. A student wants the steps left and writes a void method that stores the answer in its parameter:

ParameterAsResult.java

```
1 class StepTracker
2 {
3     private int steps;
4     private int goal;
5
6     public StepTracker(int startSteps, int dailyGoal)
7     {
8         steps = startSteps;
9         goal = dailyGoal;
10    }
11
12    public void findStepsLeft(int result)
13    {
14        result = goal - steps;
15    }
16 }
17
18 public class ParameterAsResult
19 {
20     public static void main(String[] args)
21     {
22         StepTracker tracker = new StepTracker(4200, 10000);
23         int left = 0;
24         tracker.findStepsLeft(left);
25         System.out.println(left);
26     }
27 }
```

It compiles and prints 0. Line 14 stores 5800 in `result`, but `result` is only a copy of `left`'s value. When the method ends, the copy and its 5800 vanish, and `left` still holds 0. Students expect this to work because the method "fills in" the variable it was given. The fix is to **return** the answer: make the method non-void, `public int stepsToGoal()` with `return goal - steps;`, and have the caller store it: `int left = tracker.stepsToGoal();` (as in `StepsToGoal.java` from 3.5.3). A return value is the way a method sends a primitive answer back.

Object references passed as arguments follow the same copying rule, but with an extra twist that Topic 3.6 covers.

PAUSE AND TRY

Question 11 Tracing

What does this program print?

CopyTrace.java

```
1 class Fundraiser
2 {
3     private int raised;
4
5     public Fundraiser(int startRaised)
6     {
7         raised = startRaised;
8     }
9
10    public void collect(int donation)
11    {
12        donation = donation * 2;
13        raised = raised + donation;
14    }
15
16    public int getRaised()
17    {
18        return raised;
19    }
20 }
21
22 public class CopyTrace
23 {
24     public static void main(String[] args)
25     {
26         Fundraiser drive = new Fundraiser(100);
27         int gift = 25;
28         drive.collect(gift);
29         drive.collect(gift);
30         System.out.println(gift);
31         System.out.println(drive.getRaised());
32     }
33 }
```

Question 12 Multiple choice

In `DepositCopy.java`, a student wants `cash` in `main` to be 0.0 after the deposit. Which statement is true?

- A. Line 13 already does this; `cash` is 0.0 after the call.
- B. Renaming the parameter from `amount` to `cash` would make the method change `main`'s `cash`.
- C. No statement inside `deposit` can change `main`'s `cash`; `main` itself must assign it, for example with `cash = 0.0`; after the call.
- D. Calling `bank.deposit(0)` instead would set `cash` to 0.0.

Putting it together

A phone battery class uses every kind of method from this chapter: an accessor, a void mutator, and a mutator that also returns an answer. The `use` method spends charge only if there is enough,

and it returns `true` or `false` to say whether it worked. A mutator is often void, but not always; `use` changes the object **and** reports back.

WORKED EXAMPLE A battery with three kinds of methods

BatteryDemo.java

```
1  class Battery
2  {
3      private int percent;
4
5      public Battery(int startPercent)
6      {
7          percent = startPercent;
8      }
9
10     public int getPercent()
11     {
12         return percent;
13     }
14
15     public void charge(int amount)
16     {
17         percent = percent + amount;
18         if (percent > 100)
19         {
20             percent = 100;
21         }
22     }
23
24     public boolean use(int amount)
25     {
26         if (amount > percent)
27         {
28             return false;
29         }
30         percent = percent - amount;
31         return true;
32     }
33 }
34
35 public class BatteryDemo
36 {
37     public static void main(String[] args)
38     {
39         Battery phone = new Battery(30);
40         int videoCost = 45;
41         System.out.println(phone.use(videoCost));
42         phone.charge(80);
43         System.out.println(phone.getPercent());
44         System.out.println(phone.use(videoCost));
45         System.out.println(phone.getPercent() + " " + videoCost);
46     }
47 }
```

First, classify the methods:

Method	Return type	Changes percent ?	Kind
getPercent	int	No	Accessor
charge	void	Yes	Mutator (void)
use	boolean	Sometimes	Mutator that also returns a value

Now trace `main`:

Line	Call	Inside the method	percent	Printed
39	constructor	percent set to 30	30	
41	<code>use(45)</code>	<code>45 > 30</code> is true, so line 28 returns <code>false</code> . Line 30 never runs.	30	<code>false</code>
42	<code>charge(80)</code>	<code>30 + 80 = 110</code> ; <code>110 > 100</code> , so it is capped at 100	100	
43	<code>getPercent()</code>	returns a copy, 100	100	100
44	<code>use(45)</code>	<code>45 > 100</code> is false; line 30 makes it 55; line 31 returns <code>true</code>	55	<code>true</code>
45	<code>getPercent()</code>	returns 55; joined with " " and <code>videoCost</code>	55	55 45

The output is:

```
false
100
true
55 45
```

Every idea from the chapter is visible here. `charge` is a void mutator whose call stands alone on line 42 (3.5.1, 3.5.2). `use` returns a `boolean` that `main` prints directly (3.5.3). `getPercent` reports the current value without changing it (3.5.4). On line 41, `return false`; ends `use` before the subtraction, so a failed attempt costs nothing (3.5.5). And `videoCost` is still 45 at the end, because `use` received only a copy of it (3.5.6).

Common mistakes

The Watch Out boxes explained each mistake. This table is a lookup for the compiler messages you met, so you can match an error to its cause quickly. Exact wording can vary slightly between Java versions.

Compiler message	Usual cause	Fix
invalid method declaration; return type required	A method header with no return type	Add <code>void</code> or the type the method returns
has private access in	Code in another class uses a private instance variable	Call an accessor or mutator instead
possible lossy conversion from double to int	A <code>double</code> returned from an <code>int</code> method (or passed to an <code>int</code> parameter)	Match the return type to the value

void cannot be converted to or 'void' type not allowed here	A void method's call used as a value	Call it as a statement, or write a non-void method
unexpected type (required: variable, found: value)	Assigning to a method call, such as <code>getSteps() += 500</code>	Call a mutator
unreachable statement	A statement directly after <code>return</code> in the same block	Move it above the <code>return</code>
missing return statement	Some path through a non-void method ends without a <code>return</code>	Add a <code>return</code> that covers the remaining cases

Vocabulary

Term	Definition	Example
behavior	Something an object of a class can do, written as a method in the class	Recording steps
void method	A method that does not return a value; its header has <code>void</code> before the name	<code>public void printSummary()</code>
mutator method	A method that changes an object's instance variables; often void. Also called a modifier method	<code>public void addSteps(int newSteps)</code>
parameter	A variable in a method header that receives a copy of the argument's value when the method is called	<code>newSteps</code>
non-void method	A method that returns exactly one value; its header has a return type in place of <code>void</code>	<code>public int stepsToGoal()</code>
return type	The type of value a method returns, written before its name	<code>int</code> , <code>double</code> , <code>boolean</code>
return statement	The keyword <code>return</code> , usually followed by an expression; it sends a value back and ends the method	<code>return goal - steps;</code>
return by value	Returning the value an expression evaluates to, not the variables in it	<code>return steps;</code> sends 4200
accessor method	A non-void method that returns a copy of an instance variable's value	<code>public int getSteps()</code>

Chapter summary

- Methods written inside a class give its objects behaviors. They can use the class's private instance variables by name, and a call runs the method with the instance variables of the object before the dot.

- A void method has `void` in its header and returns nothing, so its call is a statement on its own. A non-void method has a return type instead, and its return statement evaluates an expression and sends back one value, which must be compatible with the return type (return by value).
- A mutator method changes an object's instance variables, using values received through its parameters; mutators are often void. An accessor method is non-void and returns a copy of an instance variable's value, so callers can read it but not change it.
- A return statement ends the method immediately and sends control back to the caller, even from inside an `if` or a loop. Code after it does not run, and a non-void method must return a value on every path.
- A primitive argument's value is copied into its parameter. Changing the parameter never changes the caller's variable; only changes to instance variables last after the method ends.

Review questions

Question 13 Multiple choice

A `GymMembership` class has an `int` instance variable `visits` and an `int` instance variable `monthsLeft`. Its methods have these headers and behaviors:

```
1 public int getVisits()           // returns visits
2 public void checkIn()           // adds 1 to visits
3 public boolean renew(int months) // adds months to monthsLeft and returns true
4 public double costPerVisit()    // returns 40.0 / visits; changes nothing
```

Which statement is true?

- A. `renew` cannot be a mutator, because it is not a void method.
- B. `costPerVisit` is a mutator, because it uses an instance variable.
- C. `checkIn` is a mutator and `getVisits` is an accessor.
- D. A caller can change `visits` by changing the value that `getVisits()` returns.

Question 14 Tracing

A bakery sells muffins but cannot sell more than it has. What does this program print?

BakeryReview.java

```
1 class Bakery
2 {
3     private int muffins;
4
5     public Bakery(int startMuffins)
6     {
7         muffins = startMuffins;
8     }
9
10    public int sell(int wanted)
11    {
12        if (wanted > muffins)
13        {
14            wanted = muffins;
15        }
16        muffins = muffins - wanted;
```

```
17         return wanted;
18     }
19
20     public int getMuffins()
21     {
22         return muffins;
23     }
24 }
25
26 public class BakeryReview
27 {
28     public static void main(String[] args)
29     {
30         Bakery shop = new Bakery(10);
31         int order = 4;
32         int sold = shop.sell(order);
33         System.out.println(sold + " " + order);
34         order = 9;
35         sold = shop.sell(order);
36         System.out.println(sold + " " + order);
37         System.out.println(shop.getMuffins());
38     }
39 }
```

Question 15 Debugging

A movie review should earn 5 stars for a score of 90 or more, 4 stars for 70 to 89, 3 stars for 50 to 69, and 1 star for anything lower. This class does not compile. Explain the error and correct it.

StarsMissingReturn.java

```
1 class MovieReview
2 {
3     private int score;
4
5     public MovieReview(int reviewScore)
6     {
7         score = reviewScore;
8     }
9
10    public int stars()
11    {
12        if (score >= 90)
13        {
14            return 5;
15        }
16        else if (score >= 70)
17        {
18            return 4;
19        }
20        else if (score >= 50)
21        {
22            return 3;
23        }
24    }
```

```
25 }
26
27 public class StarsMissingReturn
28 {
29     public static void main(String[] args)
30     {
31         MovieReview review = new MovieReview(75);
32         System.out.println(review.stars());
33     }
34 }
```

Question 16 Tracing

What does this program print? The method changes the object's `seconds`, and it is called twice.

TimerReview.java

```
1 class CountdownTimer
2 {
3     private int seconds;
4
5     public CountdownTimer(int startSeconds)
6     {
7         seconds = startSeconds;
8     }
9
10    public int tickUntil(int alarmAt)
11    {
12        int ticks = 0;
13        while (seconds > 0)
14        {
15            seconds = seconds - 5;
16            ticks++;
17            if (seconds <= alarmAt)
18            {
19                return ticks;
20            }
21            System.out.println("tick " + seconds);
22        }
23        System.out.println("timer ended");
24        return ticks;
25    }
26
27    public int getSeconds()
28    {
29        return seconds;
30    }
31 }
32
33 public class TimerReview
34 {
35     public static void main(String[] args)
36     {
37         CountdownTimer timer = new CountdownTimer(30);
38         System.out.println(timer.tickUntil(18));
39         System.out.println(timer.getSeconds());
40     }
41 }
```

```
40         System.out.println(timer.tickUntil(-1));
41     }
42 }
```

Question 17 Multiple choice

Use the `Battery` class from Putting it together, with `Battery phone = new Battery(50);`. Which statement compiles?

- A. `int level = phone.charge(10);`
- B. `phone.getPercent() = 50;`
- C. `if (phone.use(20)) { phone.charge(5); }`
- D. `boolean ok = phone.use(12.5);`

Question 18 Short answer

An `Order` class could provide either of these methods. Both compute the same total from its `price` and `quantity` instance variables.

```
1 public void printTotal()
2 {
3     System.out.println(price * quantity);
4 }
5
6 public double getTotal()
7 {
8     return price * quantity;
9 }
```

A program needs to add a 3.00 delivery fee to the total and store the result in a variable. Which method can it use, and why can it not use the other one?

Question 19 Debugging

A gradebook should report the average of its quiz scores. After quizzes of 9 and 10 it prints 9.0 instead of 9.5. The method is only called after at least one quiz has been added. Find the error and correct it.

AverageBug.java

```
1 class Gradebook
2 {
3     private int totalPoints;
4     private int quizCount;
5
6     public Gradebook()
7     {
8         totalPoints = 0;
9         quizCount = 0;
10    }
11
12    public void addQuiz(int points)
13    {
14        totalPoints = totalPoints + points;
15        quizCount++;
16    }
17 }
```

```
16     }
17
18     public double average()
19     {
20         return totalPoints / quizCount;
21     }
22 }
23
24 public class AverageBug
25 {
26     public static void main(String[] args)
27     {
28         Gradebook book = new Gradebook();
29         book.addQuiz(9);
30         book.addQuiz(10);
31         System.out.println(book.average());
32     }
33 }
```

Reflection

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

☐ I can read a void method inside a class and explain why its call is a statement on its own.

Reread: [3.5.1 Writing a Void Method in a Class](#)

☐ I can explain what a mutator method does and trace how it changes an object over several calls. Reread: [3.5.2 Mutator Methods Change an Object](#)

☐ I can tell whether a return statement matches its method's return type and trace what the call is replaced by. Reread: [3.5.3 Non-Void Methods Return a Value](#)

☐ I can explain what an accessor method is and why changing its returned value does not change the object. Reread: [3.5.4 Accessor Methods Share a Copy](#)

☐ I can find exactly where a method stops when a return statement runs inside an if statement or a loop. Reread: [3.5.5 A Return Statement Ends the Method](#)

☐ I can explain why changing a primitive parameter never changes the caller's variable. Reread: [3.5.6 Parameters Receive Copies](#)

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. A void method returns no value, so a call to it cannot appear where a value is needed. Java reports incompatible types: void cannot be converted to int. Printing a value is not the same as returning it.

- **A** is wrong: the method prints 4200 to the screen, but printing does not hand the number back to `main`.
- **C** is wrong: "returns nothing" does not mean "returns 0". There is no value at all, so Java refuses to compile the line.
- **D** is wrong: the line in `main` never mentions `steps`. The method itself may use `steps` because it is inside the class.

Reread 3.5.1 *Writing a Void Method in a Class* if you chose differently.

Question 2

```
Steps so far: 9800
Steps so far: 4200
Steps so far: 9800
```

Lines 20 and 21 create two separate objects: `maya` with 4200 steps and `leo` with 9800. Each call runs `printSummary` with the `steps` of the object before the dot. Line 22 uses `leo`'s 9800, line 23 uses `maya`'s 4200, and line 24 uses `leo`'s 9800 again. The output follows the order of the calls, and calling a void method twice simply runs its body twice.

Reread 3.5.1 *Writing a Void Method in a Class* if your output differed.

Question 3

```
Set to 71
Set to 67
```

Line	Call	hall's temperature	office's temperature
30	constructor	68	
31	constructor	68	70
32	<code>hall.raise(4)</code>	72	70
33	<code>office.lower(3)</code>	72	67
34	<code>hall.lower(1)</code>	71	67

Each mutator call changes only the object before the dot. `office.lower(3)` has no effect on `hall`, because each object has its own `temperature`.

Reread 3.5.2 *Mutator Methods Change an Object* if your output differed.

Question 4

The error is on line 12. `total = tip;` **replaces** the total with the newest tip instead of adding to it. After the first call `total` is 2.5, and after the second it is 4.0; the starting 10.0 and the first tip are lost. The corrected line is:

```
1 total = total + tip;
```

Now the trace is 10.0 → 12.5 → 16.5, and the program prints `Tips: 16.5`. The original compiled because assigning a `double` to a `double` is legal; the mistake is in the logic, so Java cannot catch it. A mutator that accumulates must use the instance variable's old value on the right side of `=`.

Reread 3.5.2 *Mutator Methods Change an Object* if you found a different error.

Question 5

B. A `boolean` method must return a `boolean` value. `steps > goal / 2` is a relational expression, so it evaluates to `true` or `false`.

- **A** is wrong: `goal - steps` is an `int`. Java reports `incompatible types: int cannot be converted to boolean; a number is not true or false`.
- **C** is wrong: `steps / 2.0` is a `double`, which is not compatible with `boolean`.
- **D** is wrong: "yes" is a `String`, not a `boolean`. The words "yes" and `true` mean the same to a person but not to Java.

Reread 3.5.3 *Non-Void Methods Return a Value* if you chose differently.

Question 6

```
4
41.5
```

- Line 26: `order.slices()` returns $3 \times 8 = 24$, so the statement becomes `int perPerson = 24 / 5;`. Both are `ints`, so integer division gives 4 (not 4.8).
- Line 27 prints 4.
- Line 28: `order.totalCost()` returns $3 \times 12.5 = 37.5$, a `double`. The call is replaced by 37.5, and $37.5 + 4 = 41.5$ prints.

If you wrote 4.8, check the types: the returned value 24 is an `int`, and `int / int` drops the remainder. If you wrote 37.5, remember the returned value is used in the expression `+ 4` before printing.

Reread 3.5.3 *Non-Void Methods Return a Value* if your output differed.

Question 7

A. An accessor is a non-void method that returns a copy of the instance variable's value, with a return type that matches it. `balance` is a `double`, so the method returns `double`.

- **B** is wrong: it is a void method. It prints the balance on the screen, but the caller receives nothing it can store or use. An accessor must return the value.
- **C** is wrong: it assigns to `balance`, so it changes the object. That makes it a mutator, whatever its name says.
- **D** is wrong: it does not compile. Returning a `double` from an `int` method gives possible lossy conversion from `double` to `int`.

Reread 3.5.4 *Accessor Methods Share a Copy* if you chose differently.

Question 8

It prints:

```
7.0
12.0
```

Model answer: `getBalance()` returns a copy of the value in `balance`, 12.0, and line 21 stores that copy in `money`, a variable that belongs to `main`. Line 22 subtracts 5.0 from `money`, so `money` becomes 7.0. The object's `balance` is a separate variable that nothing changed, so the second call to `getBalance()` still returns 12.0.

A complete answer must:

- say that the accessor returns a copy of the value (return by value), not the instance variable itself, and
- say that changing `money` affects only `main`'s variable, so `balance` stays 12.0.

Reread 3.5.4 *Accessor Methods Share a Copy* if your answer differed.

Question 9

```
3
0
```

First call, `weeksUntil(40, 10)`, with `dollars` 15:

week at check	total	total >= 40?	Action
0	15	false	week 1, total 25
1	25	false	week 2, total 35
2	35	false	week 3, total 45
3	45	true	line 18 returns 3

Second call, `weeksUntil(10, 5)`: `total` starts at 15, and on the very first iteration `15 >= 10` is true, so line 18 returns 0 before `week` is ever increased.

In both calls the `return` inside the `if` ends the loop and the method together, so `return 10`; on line 23 never runs. Also note that line 12 creates `total` fresh on each call, starting from a copy of `dollars`; the method never changes `dollars`.

Reread 3.5.5 *A Return Statement Ends the Method* if your output differed.

Question 10

B. `95 + 10` is 105, which is greater than 100, so the `if` is true. Line 14 sets `score` to 100, and `return`; on line 15 ends the method right away. Lines 17 and 18 never run, so nothing else prints, and `getScore()` returns 100.

- **A** is wrong: it treats `return`; as if it only left the `if`. A return statement leaves the whole method, so line 18 is skipped.
- **C** is wrong: it skips the `if` entirely and runs lines 17 and 18 instead. The condition is true, so the `if` runs.
- **D** is wrong: `score` is capped at 100 by line 14, and line 17 never runs to make it 105.

Reread 3.5.5 *A Return Statement Ends the Method* if you chose differently.

Question 11

25
200

Each call to `collect` gets a fresh copy of `gift`, which is 25.

Call	donation starts	After line 12	raised after line 13	gift in main
1	25	50	150	25
2	25	50	200	25

Doubling `donation` on line 12 doubles only the copy, so `gift` stays 25 and the second call starts from 25 again, not 50. The changes to `raised`, an instance variable, last between calls.

If you answered 50 for the first line, you assumed doubling `donation` changed `gift`. If you answered 250 for the second line, you assumed the second call received 50.

Reread 3.5.6 *Parameters Receive Copies* if your output differed.

Question 12

C. A primitive argument is copied into the parameter, so the method only ever has a copy. The caller's variable can change only through an assignment in the caller.

- **A** is wrong: line 13 sets the parameter `amount`, which is a copy. The program prints `20.0` for `cash`.
- **B** is wrong: a parameter is always a new variable that belongs to the method, whatever its name. A parameter named `cash` would still be a separate copy, and `main` would still print `20.0`.
- **D** is wrong: that passes the value 0 to the method, which deposits nothing. It does not touch `cash`, which stays 20.0.

Reread 3.5.6 *Parameters Receive Copies* if you chose differently.

Question 13

C. `checkIn` changes the instance variable `visits`, which makes it a mutator. `getVisits` is non-void and returns a copy of `visits`, which makes it an accessor.

- **A** is wrong: a mutator is **often** void, not always. `renew` changes `monthsLeft`, so it is a mutator, and it also returns a `boolean`. (The battery's `use` method in Putting it together was the same kind.)
- **B** is wrong: using an instance variable is not the same as changing it. `costPerVisit` only reads `visits` to compute an answer.
- **D** is wrong: `getVisits()` returns a copy of the value. Changing the copy changes only the caller's variable.

Reread 3.5.4 *Accessor Methods Share a Copy* if you chose differently.

Question 14

4 4
6 9
0

Line	Call	wanted (copy)	Inside sell	muffins	Returned	order in main
32	sell(4)	4	4 > 10 false; 10 - 4	6	4	4
35	sell(9)	9 → 6	9 > 6 true, so wanted becomes 6; 6 - 6	0	6	9

On the second call, line 14 changes `wanted` from 9 to 6, but `wanted` is a copy, so `order in main` is still 9. The method returns the value of `wanted`, 6, which is how many muffins were actually sold. Line 37 then prints the remaining `muffins`, 0.

If you printed `6 6`, you let the change to the parameter flow back into `order`. If you printed `9 9`, you missed that the `if` lowered `wanted` before it was subtracted and returned.

Reread *3.5.6 Parameters Receive Copies* if your output differed.

Question 15

Java reports `StarsMissingReturn.java:24: error: missing return statement`, pointing at the closing brace of `stars`. If the score is below 50, all three conditions are false and control reaches the end of the method without returning anything, but the header promises an `int` on every call. It does not matter that this program only tests 75; Java checks every possible path before the program runs.

The correction is to add a return for the remaining case after the last `else if`:

```

1      else if (score >= 50)
2      {
3          return 3;
4      }
5      return 1;
6  }
```

With this fix, a review with score 75 returns 4 and one with score 32 returns 1. (Changing the last `else if (score >= 50)` into a plain `else` would compile too, but then every score below 70 would earn 3 stars, which breaks the rules above.)

Reread *3.5.5 A Return Statement Ends the Method* if you found a different error.

Question 16

```

tick 25
tick 20
3
15
tick 10
tick 5
tick 0
timer ended
3
```

First call, `tickUntil(18)`, with seconds 30:

seconds after line 15	ticks	seconds <= 18?	What happens
25	1	false	prints tick 25

20	2	false	prints tick 20
15	3	true	line 19 returns 3

The `return` ends the loop and the method, so `timer ended` does not print. `main` prints 3. `seconds` is an instance variable, so it keeps the value 15, and `getSeconds()` prints 15.

Second call, `tickUntil(-1)`: `ticks` starts at 0 again (line 12 runs on every call), but `seconds` starts at 15, where the first call left it.

seconds after line 15	ticks	seconds <= -1?	What happens
10	1	false	prints tick 10
5	2	false	prints tick 5
0	3	false	prints tick 0

Now `seconds > 0` is false, so the loop ends normally. Lines 23 and 24 run: `timer ended` prints and the method returns 3, which `main` prints.

Reread 3.5.5 *A Return Statement Ends the Method* if your output differed.

Question 17

C. `use` returns a `boolean`, so its call can serve as an `if` condition, and `charge` is `void`, so it stands alone as a statement. (With 50 percent, `use(20)` succeeds and leaves 30, and `charge(5)` makes it 35.)

- **A** is wrong: `charge` is a `void` method, so there is no value to store. Java reports `void cannot be converted to int`.
- **B** is wrong: a method call is replaced by a value, and you cannot assign to a value. Java reports `unexpected type`. Use a mutator to change the battery.
- **D** is wrong: `use` has an `int` parameter, and 12.5 is a `double`. Passing it would lose the .5, so Java reports `possible lossy conversion from double to int`.

Reread 3.5.3 *Non-Void Methods Return a Value* if you chose differently.

Question 18

Model answer: It must use `getTotal`, for example `double withFee = order.getTotal() + 3.00; .getTotal` is a non-void method, so its return statement sends the total back, and Java uses that value in place of the call, where it can be added to and stored. `printTotal` is a void method: it shows the total on the screen, but it sends nothing back to the program, so there is no value to add 3.00 to. Writing `order.printTotal() + 3.00` would not compile; Java reports `'void' type not allowed here`.

A complete answer must:

- choose `getTotal` and say that it returns a value the caller can use in an expression or store, and
- say that `printTotal` is `void`, so printing the total does not give the program a value to work with.

Reread 3.5.3 *Non-Void Methods Return a Value* if your answer differed.

Question 19

The error is on line 20. `totalPoints` and `quizCount` are both `ints`, so `totalPoints / quizCount` is integer division: `19 / 2` is 9, with the remainder thrown away. Only **then** does Java widen the

returned `int` 9 to the `double` 9.0 to match the return type. The return type being `double` does not change how the division inside the expression is done. The corrected line is:

```
1      return (double) totalPoints / quizCount;
```

The cast turns 19 into 19.0 before dividing, so the division is floating-point and the method returns 9.5. (Writing `(double) (totalPoints / quizCount)` would not help: the parentheses make the integer division happen first, giving 9.0 again.) The precondition that at least one quiz exists matters too: with `quizCount` 0, the division would be undefined.

Reread [3.5.3 Non-Void Methods Return a Value](#) if you found a different error.