

3.4 Constructors

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

By the end you will be able to

- Describe an object's state as the values of its instance variables at a given moment, and explain the has-a relationship.
- Write the header and body of a constructor that gives every instance variable a starting value.
- Explain what happens in memory when a constructor is called and what the call hands back.
- Use constructor parameters, with names different from the instance variables, to supply starting values.
- State the default values Java gives instance variables and explain when Java supplies a default constructor.
- Explain why a constructor should store a copy of a mutable object it receives instead of the object itself.

What to remember from earlier

- The arguments in a constructor call must match the parameter list in number, order, and type. ([Topic 1.13: Object Creation and Storage \(Instantiation\)](#), [1.13.6 Arguments Must Match the Parameter List](#))
- Each parameter gets a copy of its argument (call by value). ([Topic 1.13: Object Creation and Storage \(Instantiation\)](#), [1.13.7 Call by Value: Parameters Get Copies](#))
- Two reference variables holding the same reference are aliases, and == is true only for the very same object. ([Topic 2.6: Comparing Boolean Expressions](#), [2.6.4 Aliases and Reference Comparison](#))
- Each object gets its own copy of every instance variable, which is declared private. ([Topic 3.3: Anatomy of a Class](#), [3.3.2 Each Object Has Its Own Instance Variables](#))
- Constructors are always public. ([Topic 3.3: Anatomy of a Class](#), [3.3.5 Access for Constructors and Methods](#))

3.4.1 Object State and Instance Variables

Picture two library books sitting on a cart. Both are books, built from the same blueprint, but one is *Holes* with 233 pages and is checked out, and the other is *Wonder* with 310 pages and is on the shelf. If a program is going to keep track of books, it has to remember those facts separately for each one. Where does each book keep its own facts?

Recall from Topic 1.12 that the facts a class's objects keep track of are called **attributes**, and from Topic 3.3 that each attribute is stored in an **instance variable**: a variable declared inside the class but outside any constructor or method, so that every object gets its own copy. Here is a `Book` class with three attributes:

```

1 public class Book
2 {
3     private String title;
4     private int pages;
5     private boolean checkedOut;
6 }

```

Each instance variable has a type and a name, like any variable from Topic 1.2, and `private` keeps it inside the class (Topic 3.3). The class only describes *which* facts a book keeps. The values themselves live in each object.

The set of values an object's instance variables hold at a particular moment is the object's **state**. "At a particular moment" matters: if *Holes* is returned, its `checkedOut` value changes from `true` to `false`, and its state is now different, even though it is the same object.

WORKED EXAMPLE The state of two Book objects

Suppose a program has built two `Book` objects. Each one has its own three boxes:

Instance variable	State of the first book	State of the second book
<code>title</code>	"Holes"	"Wonder"
<code>pages</code>	233	310
<code>checkedOut</code>	<code>true</code>	<code>false</code>

Read down a column to get one object's state. The two objects share the same *instance variables* (the same three names and types, from the class) but have different *values*, so they have different states. Changing the first book's `checkedOut` would not touch the second book's, because each object has its own copy.

Now vary the moment instead of the object. Take a single `Thermostat` object whose instance variables are `double temperature` and `boolean heating`:

Time	<code>temperature</code>	<code>heating</code>	State
7:00 AM	17.5	<code>true</code>	cold, heater on
9:00 AM	21.0	<code>false</code>	comfortable, off

It is one object the whole time; its state at 7:00 differs from its state at 9:00 because the values changed. (The methods that change an object's state after it is built are Topic 3.5. This chapter is about the very first state an object has.)

Because the instance variables belong to the object, programmers describe the connection with the words "has a": a book **has a** title, **has a** page count, and **has a** checked-out status. This is called a **has-a relationship** between an object and its instance variables. It is a handy test when you design a class: if you can say "a _ **has a** _," the second blank is a good candidate for an instance variable.

KEY IDEA

An object's state is the set of values its instance variables hold at a given moment. The instance variables belong to the object, which gives a has-a relationship: a book has a title.

WATCH OUT

Mixing up the instance variables with the state. The instance variables (`title`, `pages`, `checkedOut`) come from the class and are the same for every `Book`. The state is the *values* in them, which differ from object to object and can change over time. "Every book has a `pages` variable" is about the class; "this book's `pages` is 233" is about one object's state.

PAUSE AND TRY**Question 1** Multiple choice

A class `Song` has the instance variables `String title`, `int seconds`, and `boolean favorite`. Which sentence describes a has-a relationship in this class?

- A. A song has a favorite status.
- B. A song has a `Song` class.
- C. A title has a song.
- D. A song has a `Book`.

Question 2 Short answer

At noon, a `Song` object holds `title = "Lift"`, `seconds = 214`, and `favorite = false`. At 3:00 the listener marks it as a favorite. Is it still the same object? Has its state changed? Explain.

3.4.2 Writing a Constructor

The `Book` class above declares three instance variables but never puts anything in them. When a program writes `new Book(...)`, which code decides that this book is *Holes* with 233 pages? In Topic 1.13 you only *called* constructors that someone else wrote. Now you write one.

A **constructor** sets the **initial state** of a new object: the values its instance variables hold at the moment it is created. A well-written constructor gives a starting value to **every** instance variable, so the object never begins life half-filled-in.

Here is the shape of a constructor, written inside the class after the instance variables:

```
1 public Book(String bookTitle, int pageCount)
2 {
3     title = bookTitle;
4     pages = pageCount;
5     checkedOut = false;
6 }
```

Piece	What it is and why it is there
<code>public</code>	Lets code in other classes call it. In this course constructors are always public (Topic 3.3)
<code>Book</code>	Exactly the class name, capital letter included; that is how Java recognizes a constructor

(nothing before the name)	No return type, not even <code>void</code> ; a constructor always produces an object of its own class
(<code>String bookTitle, int pageCount</code>)	The parameter list: the values a caller must pass in, as in Topic 1.13
<code>{ ... }</code>	The body: ordinary statements that run when the constructor is called
<code>title = bookTitle;</code> and the others	Assignments (Topic 1.4) that store a starting value in each instance variable

The body is where the instance variables get filled in. Inside a constructor you refer to an instance variable just by its name, `title`, and Java knows you mean the `title` of the object being built.

What actually happens when `new Book("Holes", 233)` runs? Three things, in order:

1. **Memory is set aside** for a brand-new `Book` object, with room for its own `title`, `pages`, and `checkedOut`.
2. **The constructor body runs** for that new object, with `bookTitle` holding a copy of "Holes" and `pageCount` a copy of 233. Its assignments fill in the instance variables.
3. **A reference to the new object is handed back** as the value of the whole `new` expression, so it can be stored in a variable.

KEY IDEA

A constructor has the class name and no return type. Calling it with `new` sets aside memory for a new object, runs the body to give every instance variable a starting value, and returns a reference to the object.

WORKED EXAMPLE Building a Book

Two classes share this file so it runs as one program. (In a real project each class goes in its own file and is marked `public`; Java allows only one `public` class per file, so the helper class here leaves the word off.) Instance variables are `private`, and writing methods that report them is Topic 3.5, so for now the constructor prints the state it just set with `println`, using string concatenation from Topic 1.15.

BookCreation.java

```

1  public class BookCreation
2  {
3      public static void main(String[] args)
4      {
5          System.out.println("Before the call");
6          Book novel = new Book("Holes", 233);
7          System.out.println("After the call");
8      }
9  }
10
11 class Book
12 {
13     private String title;
14     private int pages;
15     private boolean checkedOut;
16
17     public Book(String bookTitle, int pageCount)
18     {
19         title = bookTitle;
20         pages = pageCount;
21         checkedOut = false;
22         System.out.println(title + ", " + pages + " pages, checked out: " +
checkedOut);
23     }
24 }

```

Order	Line	What happens
1	5	Prints Before the call
2	6	new sets aside memory for a Book with its own title, pages, and checkedOut
3	17	The constructor starts: bookTitle is "Holes", pageCount is 233 (copies of the arguments)
4	19	title becomes "Holes"
5	20	pages becomes 233
6	21	checkedOut becomes false; every instance variable now has a value
7	22	Prints the new state
8	6	The constructor ends; the reference to the new object is stored in novel
9	7	Prints After the call

```

Before the call
Holes, 233 pages, checked out: false
After the call

```

Notice line 21. No parameter supplies `checkedOut`; the constructor simply decides that every new book starts on the shelf. A starting value can come from a parameter or be fixed by the constructor.

Now vary the parameter list: a constructor does not need parameters at all. Every new scoreboard starts the same way, at 0 to 0 in period 1, so its constructor takes nothing and sets fixed values. This program also builds **two** objects.

ScoreboardStart.java

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

```
Home 0, Away 0, Period 1
Home 0, Away 0, Period 1
false
```

Each `new` on lines 5 and 6 sets aside memory for a separate `Scoreboard`, runs the constructor for it, and returns a different reference. The two objects happen to have identical states, but they are two objects. Line 7 compares the references with `==` (Topic 2.6), and `false` confirms they are not aliases: changing the gym's score later would never change the field's.

WATCH OUT

Giving a constructor a return type. Because methods have a return type, students sometimes write `void` in front of a constructor's name:

BookVoid.java

```
1 public class BookVoid
2 {
3     public static void main(String[] args)
4     {
5         Book novel = new Book("Holes", 233);
6     }
7 }
8
9 class Book
10 {
11     private String title;
12     private int pages;
13
14     public void Book(String bookTitle, int pageCount)
15     {
16         title = bookTitle;
17         pages = pageCount;
18     }
19 }
```

With `void`, line 13 is no longer a constructor; it is a method that happens to be named `Book`. The class now has no constructor taking a `String` and an `int`, so line 5 fails:

```
BookVoid.java:5: error: constructor Book in class Book cannot be applied to given
types;
    required: no arguments
    found:    String,int
```

(The exact wording can vary between Java versions.) The error points at the call, but the mistake is in the header. The fix is to delete `void`:

`public Book(String bookTitle, int pageCount)`. Section 3.4.4 explains where the "no arguments" constructor in the message comes from.

PAUSE AND TRY

Question 3 Multiple choice

Which is a correct constructor header for a class named `Movie`?

- A. `public void Movie(String movieTitle)`
- B. `public Movie movie(String movieTitle)`
- C. `public Movie(String movieTitle)`
- D. `public movie(String movieTitle)`

Question 4 Tracing

What does this program print?

StudentTrace.java

```
1 public class StudentTrace
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Start");
6         Student first = new Student("Mia", 11);
7         Student second = new Student("Leo", 9);
8         System.out.println("End");
9     }
10 }
11
12 class Student
13 {
14     private String name;
15     private int grade;
16
17     public Student(String studentName, int gradeLevel)
18     {
19         name = studentName;
20         grade = gradeLevel;
21         System.out.println(name + " is in grade " + grade);
22     }
23 }
```

3.4.3 Parameters Supply the Starting Values

A constructor with no parameters builds identical objects, which is right for a scoreboard but wrong for pets: every pet has a different name and age. How does the code that writes `new Pet(...)` tell the constructor which pet to build?

Through parameters, exactly as in Topic 1.13: the caller passes arguments, and each parameter starts with a copy of its argument. The constructor's job is to move that data into the instance variables. A parameter is a temporary variable that disappears when the constructor ends; an instance variable lasts as long as the object. If the constructor does not copy a parameter's value into an instance variable, the value is lost.

There is a naming rule to follow for now: **give each parameter a different name from the instance variable it fills**, such as `petName` for `name`. You will see why in the Watch Out below.

KEY IDEA

Constructor parameters carry the data for the new object's starting state. The body must assign each one to its instance variable, because parameters vanish when the constructor ends.

WORKED EXAMPLE Two pets from one constructor

Pet has three instance variables. Two get their values from parameters; the third, `vetVisits`, always starts at 0 for a new pet.

PetCreation.java

```

1  public class PetCreation
2  {
3      public static void main(String[] args)
4      {
5          Pet dog = new Pet("Biscuit", 3);
6          Pet cat = new Pet("Pepper", 7);
7      }
8  }
9
10 class Pet
11 {
12     private String name;
13     private int age;
14     private int vetVisits;
15
16     public Pet(String petName, int petAge)
17     {
18         name = petName;
19         age = petAge;
20         vetVisits = 0;
21         System.out.println(name + ", age " + age + ", vet visits: " + vetVisits);
22     }
23 }
```

Call	petName	petAge	name set to	age set to	vetVisits set to
Line 5	"Biscuit"	3	"Biscuit"	3	0
Line 6	"Pepper"	7	"Pepper"	7	0

```

Biscuit, age 3, vet visits: 0
Pepper, age 7, vet visits: 0
```

Same constructor, different arguments, different starting states. Every instance variable got a value in both objects, two from parameters and one fixed.

Now vary what the body does with a parameter. The body is ordinary code, so it can compute a starting value from the parameters, including with a Boolean expression (Topic 2.2) and an `if` statement (Topic 2.3). A concert ticket in rows 1 through 3 is a VIP ticket costing \$80; any other row costs \$45.

ConcertTicketPrice.java

```

1  public class ConcertTicketPrice
2  {
3      public static void main(String[] args)
4      {
5          ConcertTicket front = new ConcertTicket("The Comets", 2);
6          ConcertTicket back = new ConcertTicket("The Comets", 15);
```

```
7     }
8 }
9
10 class ConcertTicket
11 {
12     private String band;
13     private int row;
14     private boolean vip;
15     private double price;
16
17     public ConcertTicket(String bandName, int seatRow)
18     {
19         band = bandName;
20         row = seatRow;
21         vip = seatRow <= 3;
22         if (vip)
23         {
24             price = 80.0;
25         }
26         else
27         {
28             price = 45.0;
29         }
30         System.out.println(band + " row " + row + ": VIP " + vip + ", $" + price);
31     }
32 }
```

For the first ticket, `seatRow` is 2, so line 21 sets `vip` to `2 <= 3`, which is `true`, and the `if` sets `price` to 80.0. For the second, `seatRow` is 15, `vip` is `false`, and the `else` sets `price` to 45.0.

```
The Comets row 2: VIP true, $80.0
The Comets row 15: VIP false, $45.0
```

The caller passed only a band and a row, yet all four instance variables got starting values: two copied from parameters and two computed from them.

WATCH OUT

Naming a parameter the same as its instance variable. It seems natural to call the parameter `name` because it fills `name`:

PetSameName.java

```
1 public class PetSameName
2 {
3     public static void main(String[] args)
4     {
5         Pet dog = new Pet("Biscuit", 3);
6         System.out.println("A Pet was created.");
7     }
8 }
9
10 class Pet
11 {
12     private String name;
13     private int age;
14
15     public Pet(String name, int age)
16     {
17         name = name;
18         age = age;
19     }
20 }
```

This compiles and runs, printing `A Pet was created.`, but the pet never gets its name or age. Inside the constructor, the name `name` now means the **parameter**, because the closer declaration wins. So line 17 copies the parameter into itself and the instance variable is never touched:

Variable	After lines 17 and 18
parameter name	"Biscuit" (copied into itself)
parameter age	3 (copied into itself)
instance variable name	still null: never assigned
instance variable age	still 0: never assigned

(Where `null` and `0` come from is section 3.4.4.) The fix is to rename the parameters, `public Pet(String petName, int petAge)`, and assign `name = petName;` and `age = petAge;`. Topic 3.9 shows a keyword that lets a parameter share the instance variable's name; until then, use different names.

PAUSE AND TRY

Question 5 Tracing

What does this program print?

GymPassTrace.java

```

1  public class GymPassTrace
2  {
3      public static void main(String[] args)
4      {
5          GymPass first = new GymPass("Ines", 3, 17);
6          GymPass second = new GymPass("Omar", 2, 25);
7      }
8  }
9
10 class GymPass
11 {
12     private String member;
13     private int months;
14     private boolean student;
15     private double cost;
16
17     public GymPass(String memberName, int monthCount, int age)
18     {
19         member = memberName;
20         months = monthCount;
21         student = age < 19;
22         cost = monthCount * 30.0;
23         if (student)
24         {
25             cost = cost / 2;
26         }
27         System.out.println(member + ": " + months + " months, $" + cost + ",
student " + student);
28     }
29 }

```

Question 6 Multiple choice

A class has the instance variable `private int speed;`. Which constructor stores the caller's value in `speed`?

- A. `public Fan(int speed) { speed = speed; }`
- B. `public Fan(int startSpeed) { startSpeed = speed; }`
- C. `public Fan(int startSpeed) { speed = startSpeed; }`
- D. `public Fan(int startSpeed) { int speed = startSpeed; }`

3.4.4 Default Values and the Default Constructor

The Watch Out above said an instance variable that was never assigned is "still `null`" or "still `0`." But nobody wrote `null` or `0` anywhere. Where do those values come from? And what happens if a class has no constructor at all?

Remember step 1 of a constructor call from 3.4.2: memory is set aside for the new object. Java never leaves that memory holding leftover junk. Before any constructor code runs, every instance variable is given a **default value** that depends only on its type:

Type of the instance variable	Default value
int	0
double	0.0
boolean	false
any reference type (String, Scanner, Book, ...)	null

Each is the "nothing yet" value for its type: zero for numbers, `false` for a true/false fact, and `null`, meaning "no object" (Topic 1.13), for references. Your constructor's assignments then replace these defaults. Any instance variable the constructor skips keeps its default.

WORKED EXAMPLE A constructor that forgets four variables

This constructor sets only `name`, then prints all five instance variables.

PlayerForgot.java

```

1  public class PlayerForgot
2  {
3      public static void main(String[] args)
4      {
5          Player rookie = new Player("Jordan");
6      }
7  }
8
9  class Player
10 {
11     private String name;
12     private String team;
13     private int goals;
14     private double average;
15     private boolean injured;
16
17     public Player(String playerName)
18     {
19         name = playerName;
20         System.out.println("name: " + name);
21         System.out.println("team: " + team);
22         System.out.println("goals: " + goals);
23         System.out.println("average: " + average);
24         System.out.println("injured: " + injured);
25     }
26 }
```

```

name: Jordan
team: null
goals: 0
average: 0.0
injured: false
```

Line 19 replaced `name`'s default. The other four were never assigned, so each still holds the default for its type: `String` gives `null`, `int` gives `0`, `double` gives `0.0`, and `boolean` gives `false`. This program runs without an error, which is exactly the danger: `team` being `null` may cause

trouble much later, far from this constructor. That is why a constructor should set **every** instance variable on purpose, even when the value it wants happens to match the default.

Now vary the class: remove the constructor entirely.

DefaultLamp.java

```
1 public class DefaultLamp
2 {
3     public static void main(String[] args)
4     {
5         Lamp desk = new Lamp();
6         System.out.println("A Lamp was created.");
7     }
8 }
9
10 class Lamp
11 {
12     private String color;
13     private int brightness;
14     private double watts;
15     private boolean on;
16 }
```

A Lamp was created.

Lamp has no constructor, yet line 5 calls `new Lamp()` and it works. When a class has **no constructor written at all**, Java supplies one: a constructor with no parameters and an empty body. It is called the **default constructor**. Since its body does nothing, the new lamp's instance variables keep their defaults: `color` is `null`, `brightness` is `0`, `watts` is `0.0`, and `on` is `false`.

KEY IDEA

Instance variables start with default values: `0` for `int`, `0.0` for `double`, `false` for `boolean`, and `null` for reference types. If a class has no constructor written, Java provides a no-parameter default constructor that leaves every instance variable at its default.

WATCH OUT

Expecting the default constructor after you write your own. Java supplies the default constructor only when the class has *no* constructor. As soon as you write any constructor, Java stops supplying it:

NoDefaultAnymore.java

```
1 public class NoDefaultAnymore
2 {
3     public static void main(String[] args)
4     {
5         Lamp desk = new Lamp();
6     }
7 }
8
9 class Lamp
10 {
11     private String color;
12     private int brightness;
13
14     public Lamp(String lampColor, int level)
15     {
16         color = lampColor;
17         brightness = level;
18     }
19 }
```

```
NoDefaultAnymore.java:5: error: constructor Lamp in class Lamp cannot be applied to
given types;
    required: String,int
    found:    no arguments
    reason: actual and formal argument lists differ in length
```

The only constructor needs a `String` and an `int`, and the call passes nothing. Fix it by passing the arguments the constructor needs, such as `new Lamp("white", 60)`, or, if the class should also allow `new Lamp()`, by writing a no-parameter constructor too (two constructors with different signatures are overloaded, Topic 1.13). This also explains the `BookVoid` error in 3.4.2: with the "constructor" turned into a method, `Book` had no constructor, so Java supplied the default one, which takes no arguments.

PAUSE AND TRY**Question 7** Tracing

What does this program print?

SensorDefaults.java

```
1 public class SensorDefaults
2 {
3     public static void main(String[] args)
4     {
5         Sensor porch = new Sensor("porch", 21.5);
6     }
7 }
8
9 class Sensor
10 {
11     private String location;
12     private double temperature;
13     private int readings;
14     private boolean alarm;
15     private String unit;
16
17     public Sensor(String place, double startTemp)
18     {
19         location = place;
20         readings = readings + 1;
21         System.out.println(location + " " + temperature + " " + readings);
22         System.out.println(alarm + " " + unit);
23     }
24 }
```

Question 8 Debugging

This program should print `Holes has 233 pages`, but it prints `Holes has 0 pages`. Find the error, explain why `0` appears, and write the corrected line.

BookReversed.java

```
1 public class BookReversed
2 {
3     public static void main(String[] args)
4     {
5         Book story = new Book("Holes", 233);
6     }
7 }
8
9 class Book
10 {
11     private String title;
12     private int pages;
13
14     public Book(String bookTitle, int pageCount)
15     {
16         title = bookTitle;
17         pageCount = pages;
18         System.out.println(title + " has " + pages + " pages");
19     }
20 }
```

3.4.5 Mutable Objects as Parameters

So far every parameter has been a primitive or a `String`. What if a constructor receives an object that can *change*? Call by value copies the argument, but for a reference type the argument is a reference, so the constructor gets a copy of the **address**, not a copy of the object.

An object whose state can change after it is built is called **mutable**. You already know one. In Topic 2.6 you saw that a `Scanner` moves forward through its input each time you read from it, so reading changes its state. A `String` is not mutable: once built, its characters never change (Topic 1.15). Storing a reference to a `String` is always safe, because nobody can change the object behind your back.

WORKED EXAMPLE A grader that shares the caller's `Scanner`

The `Grader` constructor stores the `Scanner` it receives and reads the first answer from it.

GraderShared.java

```
1 import java.util.Scanner;
2
3 public class GraderShared
4 {
5     public static void main(String[] args)
6     {
7         Scanner key = new Scanner("B D A C");
8         Grader quizOne = new Grader(key);
9         System.out.println("main reads: " + key.next());
10    }
11 }
12
13 class Grader
14 {
15     private Scanner answers;
16     private String firstAnswer;
17
18     public Grader(Scanner answerKey)
19     {
20         answers = answerKey;
21         firstAnswer = answers.next();
22         System.out.println("Grader reads: " + firstAnswer);
23     }
24 }
```

```
Grader reads: B
main reads: D
```

Line 20 stores the parameter's reference in the instance variable, so `answers` (inside the object) and `key` (in `main`) are aliases of **one** `Scanner`. Line 21 reads `B` through `answers`, and that moves the shared `Scanner` forward. When `main` reads through `key` on line 9, it gets `D`, not `B`. Building a `Grader` quietly changed an object that belongs to `main`. The reverse is also true: anything `main` later does with `key` changes what the grader will read next.

This is why, **when a constructor receives a mutable object, the instance variable should hold a copy of that object**: a new object with the same values. Then the instance variable does not refer

to the caller's object, so code in the class cannot change the caller's object, and changes the caller makes cannot reach inside the new object.

A `Scanner` cannot be copied, but a class can offer a way to copy its own objects. Below, `Address` is mutable: its `moveTo` method changes the street. (Lines 27–30 are a method; you will learn to write methods in Topic 3.5. You only need to know that calling `moveTo` changes an `Address` object's street.) `Address` also has a second constructor, on lines 21–25, that builds a new `Address` holding the same street and city as one it is given. It can read `original.street` because `private` limits access to code written inside the `Address` class (Topic 3.3), and this constructor is inside it.

`OrderCopy.java`

```
1 public class OrderCopy
2 {
3     public static void main(String[] args)
4     {
5         Address home = new Address("12 Oak St", "Akron");
6         Order lamp = new Order("lamp", home);
7     }
8 }
9
10 class Address
11 {
12     private String street;
13     private String city;
14
15     public Address(String streetName, String cityName)
16     {
17         street = streetName;
18         city = cityName;
19     }
20
21     public Address(Address original)
22     {
23         street = original.street;
24         city = original.city;
25     }
26
27     public void moveTo(String newStreet)
28     {
29         street = newStreet;
30     }
31 }
32
33 class Order
34 {
35     private String item;
36     private Address shipTo;
37
38     public Order(String itemName, Address destination)
39     {
40         item = itemName;
41         shipTo = new Address(destination);
42         System.out.println("Same Address object as the caller's? " + (shipTo ==
43 destination));
```

```
43     }
44 }
```

Same Address object as the caller's? false

Line 41 calls `new Address(destination)`, so `shipTo` gets a reference to a **new** object with the same values. Line 42 compares references with `==` and confirms that `shipTo` and the caller's `home` are different objects. If the order had instead stored the reference directly, `shipTo = destination;`, the same program prints `Same Address object as the caller's? true:`

Line 41 is	What <code>shipTo</code> refers to	If main later calls <code>home.moveTo("9 Elm St")</code>
<code>shipTo = destination;</code>	the caller's own object	the order's address changes too
<code>shipTo = new Address(destination);</code>	a new copy	the order still ships to 12 Oak St

KEY IDEA

When a constructor receives a mutable object, initialize the instance variable with a copy of that object, not the reference itself. Otherwise the object and the caller share one object, and a change through either one shows up in both.

WATCH OUT

Thinking call by value already protects objects. Call by value copies the *argument*, and for a reference type the argument is a reference. `shipTo = destination;` copies an address onto a second sticky note; it does not build a second house. Only `new` builds a new object, so making a real copy always involves `new`.

PAUSE AND TRY

Question 9 Multiple choice

A class `Team` has the instance variable `private Roster players;`, where `Roster` is a mutable class with a constructor `Roster(Roster original)` that makes a copy. Which line in `Team`'s constructor follows the guideline for a parameter `Roster startPlayers`?

- A. `players = startPlayers;`
- B. `startPlayers = players;`
- C. `players = new Roster(startPlayers);`
- D. `Roster players = startPlayers;`

Question 10 Short answer

In `GraderShared.java`, a classmate says, "`main` passed key by value, so the `Grader` got its own copy and reading from it cannot affect `main`." Explain what is wrong with this reasoning.

Putting it together

A hotel program records each stay. A stay has a guest, a number of nights, a total cost, a paid status, and a billing address. Guests booking just one night can leave out the number of nights. Read the whole file, then follow the trace.

Booking.java

```
1 public class Booking
2 {
3     public static void main(String[] args)
4     {
5         Address ana = new Address("12 Oak St", "Akron");
6         System.out.println("Booking rooms...");
7         HotelStay longStay = new HotelStay("Ana", 3, ana);
8         HotelStay quickStay = new HotelStay("Ana", ana);
9         System.out.println("Done");
10    }
11 }
12
13 class Address
14 {
15     private String street;
16     private String city;
17
18     public Address(String streetName, String cityName)
19     {
20         street = streetName;
21         city = cityName;
22     }
23
24     public Address(Address original)
25     {
26         street = original.street;
27         city = original.city;
28     }
29
30     public void moveTo(String newStreet)
31     {
32         street = newStreet;
33     }
34 }
35
36 class HotelStay
37 {
38     private String guest;
39     private int nights;
40     private double total;
41     private boolean paid;
42     private Address billing;
43
44     public HotelStay(String guestName, int stayNights, Address guestAddress)
45     {
46         guest = guestName;
47         nights = stayNights;
48         total = stayNights * 89.5;
49         paid = false;
50         billing = new Address(guestAddress);
```

```

51         System.out.println(guest + ": " + nights + " nights, $" + total + ", paid "
+ paid);
52     }
53
54     public HotelStay(String guestName, Address guestAddress)
55     {
56         guest = guestName;
57         nights = 1;
58         total = 89.5;
59         paid = false;
60         billing = new Address(guestAddress);
61         System.out.println(guest + ": " + nights + " night, $" + total + ", paid " +
paid);
62     }
63 }

```

Order	Line(s)	What happens	Printed
1	5	An Address is built for 12 Oak St, Akron; ana refers to it	
2	6	Prints	Booking rooms...
3	7	Three arguments match HotelStay(String, int, Address); memory is set aside with defaults	
4	46-49	guest is "Ana", nights is 3, total is 3 * 89.5 = 268.5, paid is false	
5	50	new Address(guestAddress) builds a copy, so billing does not share main's Address	
6	51	Prints the state; the reference is stored in longStay	Ana: 3 nights, \$268.5, paid false
7	8	Two arguments match HotelStay(String, Address), the overloaded constructor	
8	56-60	nights is fixed at 1, total at 89.5, paid at false; billing is another copy	
9	61	Prints the state; the reference is stored in quickStay	Ana: 1 night, \$89.5, paid false
10	9	Prints	Done

```

Booking rooms...
Ana: 3 nights, $268.5, paid false
Ana: 1 night, $89.5, paid false
Done

```

Check the habits from this chapter against the code. Both constructors set all five instance variables, including `paid = false`, which matches the default but is written on purpose. Every parameter name differs from its instance variable. And because `Address` is mutable, each stay keeps its own copy: if Ana later moves and `main` calls `ana.moveTo(...)`, the addresses on the

existing stays do not change. Five objects exist at the end, one for each `new` that ran (lines 5, 7, 50, 8, and 60): `ana`'s `Address`, two `HotelStay` objects, and two `Address` copies.

Vocabulary

Term	Definition	Example
attribute	A fact an object keeps track of	a book's title
instance variable	A variable declared in a class, outside constructors and methods, that each object has its own copy of	<code>private int pages;</code>
state	The values of an object's instance variables at a given moment	<code>"Holes", 233, false</code>
has-a relationship	The connection between an object and its instance variables	a book has a title
constructor	Code with the class name and no return type that sets a new object's initial state	<code>public Book(String bookTitle, int pageCount)</code>
initial state	The values an object's instance variables hold when it is created	a new scoreboard at 0 to 0
default value	The value an instance variable has before any code assigns it, based on its type	<code>0, 0.0, false, null</code>
default constructor	The no-parameter constructor Java supplies when a class has no constructor written	<code>new Lamp()</code> with no constructor in <code>Lamp</code>
mutable	Describes an object whose state can change after it is built	a <code>Scanner</code> ; not a <code>String</code>

Chapter summary

- An object's **state** is the values of its instance variables at a given moment. The instance variables belong to the object, giving a **has-a relationship**, and each object has its own copy.
- A **constructor** has the class name, is `public`, and has no return type. Calling it with `new` sets aside memory for the object, runs the body, and returns a reference to the new object.
- A constructor should give every instance variable a starting value. Values can come from parameters (named differently from the instance variables), be fixed, or be computed from parameters.
- Instance variables start at their **default values** (`0, 0.0, false, null`). If a class has no constructor, Java supplies a no-parameter **default constructor** that leaves them that way; writing any constructor removes it.
- When a constructor receives a **mutable** object, the instance variable should hold a copy made with `new`, so the object and the caller never share it.

Review questions

Question 11 Multiple choice

Book has the constructor from 3.4.2. Which list gives, in order, what happens when `Book novel = new Book("Holes", 233);` runs?

- A. The body runs → memory is set aside → the reference is stored in `novel`
- B. Memory is set aside → the body runs → the reference is stored in `novel`
- C. The reference is stored in `novel` → memory is set aside → the body runs
- D. Memory is set aside → the reference is stored in `novel` → the body runs

Question 12 Short answer

A bike-share program needs a `Bike` class. Each bike has an ID number, a battery level from 0.0 to 100.0, and whether it is currently rented. Name a type for each instance variable, explain the has-a relationship, and describe what a reasonable constructor that takes only the ID would set the initial state to.

Question 13 Tracing

What does this program print?

CampSignup.java

```
1 public class CampSignup
2 {
3     public static void main(String[] args)
4     {
5         Camper first = new Camper("Rosa", 12);
6         Camper second = new Camper("Eli", 8);
7         Camper third = new Camper("Kai");
8     }
9 }
10
11 class Camper
12 {
13     private String name;
14     private int age;
15     private boolean swimmer;
16     private double fee;
17
18     public Camper(String camperName, int camperAge)
19     {
20         name = camperName;
21         age = camperAge;
22         swimmer = camperAge >= 10;
23         fee = 150.0;
24         System.out.println(name + " " + age + " " + swimmer + " " + fee);
25     }
26
27     public Camper(String camperName)
28     {
29         name = camperName;
30         System.out.println(name + " " + age + " " + swimmer + " " + fee);
31     }
32 }
```

Question 14 Debugging

This program should create locker 214 with a blue lock, but it does not compile. Explain the error and write a corrected line 5.

LockerNoArgs.java

```
1 public class LockerNoArgs
2 {
3     public static void main(String[] args)
4     {
5         Locker gymLocker = new Locker();
6     }
7 }
8
9 class Locker
10 {
11     private int number;
12     private String lockColor;
13
14     public Locker(int lockerNumber, String color)
15     {
16         number = lockerNumber;
17         lockColor = color;
18         System.out.println("Locker " + number + " has a " + lockColor + " lock");
19     }
20 }
```

Question 15 Multiple choice

A class has the instance variables `private String city;`, `private double rainfall;`, and `private boolean flooded;`. Which constructor body correctly sets the whole initial state, given the parameters `String cityName` and `double inches`?

- A. `city = cityName; rainfall = inches;`
- B. `cityName = city; inches = rainfall; flooded = false;`
- C. `city = cityName; rainfall = inches; flooded = inches > 10.0;`
- D. `String city = cityName; double rainfall = inches; boolean flooded = false;`

Question 16 Short answer

The `Order` constructor in 3.4.5 makes a copy of its `Address` parameter, but the `Book` constructor in 3.4.2 stores its `String` parameter directly with `title = bookTitle;`. Explain why `Book` does not need to copy the `String`.

Question 17 Tracing

This program uses the `Grader` class from 3.4.5. What does it print? Then explain in one sentence why the second grader does not read A.

GraderTwice.java

```
1 import java.util.Scanner;
2
3 public class GraderTwice
4 {
5     public static void main(String[] args)
6     {
7         Scanner key = new Scanner("A C B D");
8         Grader quizOne = new Grader(key);
9         Grader quizTwo = new Grader(key);
10        System.out.println("main reads: " + key.next());
11    }
12 }
13
14 class Grader
15 {
16     private Scanner answers;
17     private String firstAnswer;
18
19     public Grader(Scanner answerKey)
20     {
21         answers = answerKey;
22         firstAnswer = answers.next();
23         System.out.println("Grader reads: " + firstAnswer);
24     }
25 }
```

Reflection

- ☐ I can describe an object's state and the has-a relationship between an object and its instance variables. Reread: 3.4.1 Object State and Instance Variables
- ☐ I can write a constructor header with no return type and explain what happens in memory when it is called. Reread: 3.4.2 Writing a Constructor
- ☐ I can use parameters with different names to give every instance variable a starting value. Reread: 3.4.3 Parameters Supply the Starting Values
- ☐ I can state the default values for int, double, boolean, and reference types, and say when Java supplies a default constructor. Reread: 3.4.4 Default Values and the Default Constructor
- ☐ I can explain why a constructor should copy a mutable object it receives. Reread: 3.4.5 Mutable Objects as Parameters

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

A. In a has-a relationship, the object has each of its instance variables. `favorite` is an instance variable of `Song`, so "a song has a favorite status" fits.

- **B** is wrong: the class is the blueprint the object is built from, not something stored inside the object.
- **C** is wrong: it reverses the relationship. The object has the attribute, not the other way around.
- **D** is wrong: `Song` has no instance variable for a book.

Reread 3.4.1 *Object State and Instance Variables* if you chose differently.

Question 2

Model answer: It is the same object, because no new object was built; one of its instance variables just got a new value. Its state has changed: at noon the values were `"Lift"`, `214`, `false`, and at 3:00 they are `"Lift"`, `214`, `true`. State means the values at a given moment, so changing any one value changes the state.

A complete answer must:

- say it is the same object, and
- say the state changed because `favorite` now holds a different value.

Reread 3.4.1 *Object State and Instance Variables* if your answer differed.

Question 3

C. A constructor is `public`, has exactly the class name, and has no return type.

- **A** is wrong: `void` is a return type, so this declares a method named `Movie`, not a constructor.
- **B** is wrong: `Movie` here is a return type in front of a method named `movie`, so it is a method too.
- **D** is wrong: the name must match the class name exactly, including the capital `M`.

Reread 3.4.2 *Writing a Constructor* if you chose differently.

Question 4

```
Start
Mia is in grade 11
Leo is in grade 9
End
```

Order	Line(s)	What happens
1	5	Prints <code>Start</code>
2	6, 17–21	First object: name is <code>"Mia"</code> , grade is <code>11</code> ; prints its state
3	6	Reference to the first object stored in <code>first</code>
4	7, 17–21	Second object, separate memory: name is <code>"Leo"</code> , grade is <code>9</code> ; prints it
5	7	Reference to the second object stored in <code>second</code>
6	8	Prints <code>End</code>

Each `new` runs the constructor completely before `main` continues, and each object gets its own instance variables, so setting Leo's values does not change Mia's.

Reread 3.4.2 *Writing a Constructor* if your output differed.

Question 5

```
Ines: 3 months, $45.0, student true
Omar: 2 months, $60.0, student false
```

Line	First call ("Ines", 3, 17)	Second call ("Omar", 2, 25)
19	<code>member</code> is "Ines"	<code>member</code> is "Omar"
20	<code>months</code> is 3	<code>months</code> is 2
21	<code>17 < 19</code> : <code>student</code> is true	<code>25 < 19</code> : <code>student</code> is false
22	<code>cost</code> is <code>3 * 30.0 = 90.0</code>	<code>cost</code> is <code>2 * 30.0 = 60.0</code>
23–26	<code>student</code> is true: <code>cost</code> becomes 45.0	skipped

`age` is a parameter only; no instance variable stores it. It is used to compute `student` and then disappears when the constructor ends.

Reread 3.4.3 *Parameters Supply the Starting Values* if your output differed.

Question 6

C. The parameter has a different name, and the assignment puts the parameter's value into the instance variable (the variable being changed goes on the left of `=`).

- **A** is wrong: inside the constructor `speed` means the parameter, so this copies the parameter into itself and the instance variable is never assigned.
- **B** is wrong: the assignment runs backward. It overwrites the parameter with the instance variable's value, and `speed` is never set.
- **D** is wrong: `int speed` declares a brand-new local variable that disappears when the constructor ends; the instance variable is never set.

Reread 3.4.3 *Parameters Supply the Starting Values* if you chose differently.

Question 7

```
porch 0.0 1
false null
```

Instance variable	Default	After the constructor body
<code>location</code>	<code>null</code>	"porch" (line 19)
<code>temperature</code>	<code>0.0</code>	still <code>0.0</code> : <code>startTemp</code> is never stored
<code>readings</code>	<code>0</code>	<code>0 + 1 = 1</code> (line 20 starts from the default)
<code>alarm</code>	<code>false</code>	still <code>false</code>
<code>unit</code>	<code>null</code>	still <code>null</code>

The trap is `temperature`: the caller passed 21.5, but the constructor never assigns `startTemp` to anything, so the value is lost when the constructor ends.

Reread 3.4.4 *Default Values and the Default Constructor* if your output differed.

Question 8

Corrected line 17:

```
1 pages = pageCount;
```

Line 17 has the assignment backward. `=` stores the right side into the variable on the left, so `pageCount = pages;` overwrites the parameter with the instance variable's value and never changes `pages`. `pages` therefore keeps its default, `0` for an `int`, and line 18 prints `Holes` has `0` pages. There is no compile error because both sides are `int` variables. With the fix, the program prints `Holes` has `233` pages.

Reread 3.4.3 *Parameters Supply the Starting Values* and 3.4.4 *Default Values and the Default Constructor* if you missed it.

Question 9

C. For a mutable parameter, the instance variable should refer to a copy, and `new Roster(startPlayers)` builds a new `Roster` with the same values.

- **A** is wrong: it stores the caller's reference, so the team and the caller share one `Roster`; a change through either shows up in both.
- **B** is wrong: it runs backward and changes only the parameter; `players` keeps its default, `null`.
- **D** is wrong: it declares a new local variable named `players` that disappears when the constructor ends, and it would share the caller's object anyway.

Reread 3.4.5 *Mutable Objects as Parameters* if you chose differently.

Question 10

Model answer: Call by value copies the argument, but `key` holds a reference, so the constructor received a copy of the reference, not a copy of the `Scanner`. Line 20 stores that reference, making `answers` and `key` aliases of one `Scanner` object. A `Scanner` is mutable, so when the grader reads `B`, the shared object moves forward and `main` reads `D` next.

A complete answer must:

- say the copy is of the reference (address), not of the object, and
- explain that both variables then refer to one mutable object, so a change through one is seen through the other.

Reread 3.4.5 *Mutable Objects as Parameters* if your reasoning differed.

Question 11

B. `new` first sets aside memory for the object (instance variables at their defaults), then the constructor body runs to set the initial state, and only when it finishes is the reference returned and stored by `=`.

- **A** is wrong: the body assigns to the object's instance variables, so the object's memory must exist before the body can run.
- **C** is wrong: there is no reference to store until the object exists, and `=` stores the result of the whole `new` expression, which is ready only after the constructor finishes.
- **D** is wrong: the reference is stored only after the constructor finishes; Topic 1.13's flow of control shows the call completes before the statement does.

Reread 3.4.2 *Writing a Constructor* if you chose differently.

Question 12

Model answer: `int id`, `double battery`, and `boolean rented`. A bike has an ID, has a battery level, and has a rented status, so these are instance variables each `Bike` object owns. A constructor with one `int` parameter (for example `bikeId`) would set `id` from the parameter and give the others fixed starting values, such as `battery = 100.0`; and `rented = false`; for a new, charged bike that is not yet rented. It should set all three on purpose rather than rely on defaults; a default battery of `0.0` would mean a dead bike.

A complete answer must:

- give sensible types (`int`, `double`, `boolean`),
- describe the has-a relationship, and
- say the constructor sets every instance variable, one from the parameter and the rest to chosen starting values.

Reread 3.4.1 *Object State and Instance Variables* and 3.4.3 *Parameters Supply the Starting Values* if your answer differed.

Question 13

```
Rosa 12 true 150.0
Eli 8 false 150.0
Kai 0 false 0.0
```

Call	Constructor	name	age	swimmer	fee
<code>new Camper("Rosa", 12)</code>	<code>Camper(String, int)</code>	"Rosa"	12	<code>12 >= 10: true</code>	150.0
<code>new Camper("Eli", 8)</code>	<code>Camper(String, int)</code>	"Eli"	8	<code>8 >= 10: false</code>	150.0
<code>new Camper("Kai")</code>	<code>Camper(String)</code>	"Kai"	0	<code>false (default)</code>	<code>0.0 (default)</code>

The one-argument constructor sets only `name`, so the other three keep their defaults. That is legal, but it is why constructors should set every instance variable: Kai's fee of `0.0` is almost certainly a bug.

Reread 3.4.4 *Default Values and the Default Constructor* if your third line differed.

Question 14

Corrected line 5:

```
1 Locker gymLocker = new Locker(214, "blue");
```

The compiler reports constructor `Locker` in class `Locker` cannot be applied to given types with required: `int,String` and found: no arguments. `Locker` has a constructor that needs an `int` and a `String`. Because the class has a constructor written, Java does not supply a default constructor, so there is no `Locker()` to call. Passing the locker number and lock color matches the parameter list in number, order, and type. The fixed program prints `Locker 214 has a blue lock`.

Reread 3.4.4 *Default Values and the Default Constructor* if you expected `new Locker()` to work.

Question 15

C. Each instance variable is on the left of `=` and gets a value: two from the parameters and `flooded` computed from `inches`.

- **A** is wrong: it compiles, but `flooded` is never set and only has the default `false` by accident. A constructor should set every instance variable on purpose.
- **B** is wrong: the first two assignments run backward, overwriting the parameters, so `city` stays `null` and `rainfall` stays `0.0`.
- **D** is wrong: each line declares a new local variable that disappears when the constructor ends, so all three instance variables keep their defaults.

Reread 3.4.3 *Parameters Supply the Starting Values* if you chose differently.

Question 16

Model answer: The copy rule is for mutable objects. A `String` is not mutable: once it is built, its characters can never change. `title` and the caller's variable do refer to the same `String` object, but since nobody can change that object, sharing it cannot cause a surprise. An `Address` can change through `moveTo`, so sharing it would let a change by the caller show up in the order.

A complete answer must:

- say `String` objects cannot change (are not mutable) while `Address` objects can, and
- explain that sharing only causes trouble when the shared object can change.

Reread 3.4.5 *Mutable Objects as Parameters* if your reasoning differed.

Question 17

```
Grader reads: A
Grader reads: C
main reads: B
```

Step	Read through	Token read	Next token in the shared Scanner
Line 8, first grader	answers (alias of key)	A	C
Line 9, second grader	answers (alias of key)	C	B
Line 10, main	key	B	D

Both graders store the reference they were given, so `key`, the first grader's `answers`, and the second grader's `answers` are all aliases of one mutable `Scanner`. Each read moves that one object forward, so the second grader reads `C` instead of starting from `A`.

Reread 3.4.5 *Mutable Objects as Parameters* if your output differed.