

3.1 Abstraction and Program Design

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

By the end you will be able to

- Explain abstraction as keeping the details that matter for a problem and hiding the rest.
- Explain how data abstraction lets you think about data by its name and meaning instead of how it is stored.
- Describe the attributes of a class and tell instance variables (one value per object) from class variables (one value shared by every object).
- Explain procedural abstraction and break a large behavior into smaller methods.
- Explain how parameters let one method replace several copies of nearly identical code.
- Decide whether a change to a method's inside can be made without telling the programmers who call it.
- Plan a class before coding it by listing its attributes and behaviors in words or in a class diagram.

What to remember from earlier

- A method is a named block of code that runs when it is called. ([Topic 1.9: Method Signatures, 1.9.1 Methods Are Named Blocks of Code](#))
- Procedural abstraction lets you use a method by knowing what it does, not how it was written. ([Topic 1.9: Method Signatures, 1.9.2 Procedural Abstraction](#))
- A signature is the method name plus the list of its parameter types. ([Topic 1.9: Method Signatures, 1.9.5 Method Signatures](#))
- A class is a blueprint for attributes and behaviors, and each object is an instance with its own values. ([Topic 1.12: Objects: Instances of Classes, 1.12.1 Classes and Objects](#))
- A constructor is called with new to create an object. ([Topic 1.13: Object Creation and Storage \(Instantiation\), 1.13.1 Constructors and the new Keyword](#))
- A statement execution count is how many times a statement runs. ([Topic 2.12: Informal Run-Time Analysis, 2.12.1 Counting How Many Times a Statement Runs](#))

3.1.1 Abstraction: Keeping What Matters

Imagine your school asks you to write a program that tracks library books. A real book has hundreds of facts about it: its title, author, number of pages, cover color, weight, the font it is printed in, the city where it was printed, and whether someone spilled juice on page 40. If your program tried to store every one of those facts, it would be enormous, slow to write, and confusing to read. Yet most of those facts have nothing to do with lending books.

Good programmers handle this the same way a good mapmaker does. A subway map leaves out streets, parks, and buildings and keeps only the stations and the lines between them, because a rider only needs to know where to get on and off. A hiking map of the same city keeps the trails and hills and leaves out the subway. Neither map is "missing" information; each one keeps exactly what its reader needs.

That habit has a name. **Abstraction** means making a problem simpler by concentrating on its central idea: you hide the details that do not matter for the question you are answering, and you gather together the details that do. The word "abstract" here means "pulled away from the messy specifics." What counts as a detail that matters always depends on the question at hand.

KEY IDEA

Abstraction reduces complexity by hiding details that do not matter for the problem and bringing together the details that do. Which details matter depends on what the program is for.

WORKED EXAMPLE Choosing the details for a library program

The library program needs to answer questions like "Is this book on the shelf?", "Who has it?", and "When is it due back?" Go through some facts about a book and ask of each one, "Does the program need this to answer its questions?"

Fact about a book	Keep or hide?	Reason
Title	Keep	Students search for books by title
Author	Keep	Students also search by author
Whether it is checked out	Keep	Needed to answer "Is it on the shelf?"
Due date	Keep	Needed to answer "When is it due back?"
Cover color	Hide	No question the library asks depends on it
Weight in grams	Hide	Lending a book does not depend on how heavy it is
City where it was printed	Hide	Interesting, but no lending decision uses it

Four facts out of hundreds. The program now has a small, clear picture of a book: the main idea of a library book, as far as lending goes. Every fact that was hidden still exists in the real world; the program simply does not need to think about it.

Now change one thing: the question. Suppose a shipping company writes a program for mailing books to stores. For that program, **weight is essential**, because postage depends on it, and the size of the book matters for choosing a box. The due date and "checked out" status are now useless. The same real object, a book, gets a different abstraction because the program is answering different questions. There is no single correct list of details for "a book"; there is only the right list for a particular problem.

WATCH OUT

"Abstraction means leaving things out to make it simpler, so leave out as much as possible." A student designing the library program decides to keep only the title, because "simpler is better." Now the program cannot answer "When is it due back?" at all, because the due date was hidden.

Abstraction is not about removing details; it is about choosing them. Hiding a detail the program needs breaks the program, just as keeping hundreds of details it does not need buries the important ones. For every detail, ask "Does any question this program answers depend on it?" Keep it if yes, hide it if no.

PAUSE AND TRY**Question 1** Multiple choice

A city is writing a program that tells riders when the next bus will reach their stop. Which set of details about a bus best fits this program?

- A. Paint color, number of seats, and the driver's favorite radio station
- B. Route number, current location, and the list of stops on its route
- C. Every fact the city has ever recorded about the bus, to be safe
- D. Route number only, since riders only care which bus it is

Question 2 Short answer

A school nurse's program and a school bus-routing program both keep information about students. Name one detail that the nurse's program should keep but the bus program should hide, and one detail that the bus program should keep but the nurse's program should hide. Explain how this shows what abstraction depends on.

3.1.2 Data Abstraction: Naming Data

You have been using abstraction on data since your first program. When you wrote `int score = 92`; in Topic 1.2, you thought of `score` as "the score," not as a pattern of 1s and 0s stored somewhere in memory. When you called `length()` on a `String` in Topic 1.15, you got the number of characters without ever learning how Java stores those characters. You used the data through its name and its meaning, and the storage details stayed out of your way.

That is **data abstraction**: separating what a piece of data *means* (its abstract properties) from the specific way it is *stored* (its concrete **representation**). Data abstraction manages complexity by giving the data a name you can think and talk about, without mentioning how it is represented. A person reading `startTime` in a program thinks "the time the event starts," not "an `int` holding minutes since midnight."

Data abstraction works at different sizes:

- **A single variable.** `double temperature` names one value.
- **A class.** A `String` brings many characters together under one name, so you can think about "the word `"hopper"`" instead of six separate characters. A class that you design can bring together several related pieces of data, as you will see in 3.1.3.
- **A set of data.** "This year's daily temperatures" names a whole collection of values as one thing. You will learn ways to store sets of data in Unit 4; the abstraction, thinking of the whole collection by one name, already makes sense.

KEY IDEA

Data abstraction separates what data means from how it is stored. You work with data through a name and its meaning, and the details of its representation stay hidden.

WORKED EXAMPLE One idea, two representations

A school schedule needs to store the time a class starts: 10:15 in the morning. Here are two ways to represent that same time. Look at how differently they are stored, and then at what each one prints.

```
1 int startMinutes = 615;
2 System.out.println("Starts at " + startMinutes / 60 + ":" + startMinutes % 60);
3
4 int startHour = 10;
5 int startMinute = 15;
6 System.out.println("Starts at " + startHour + ":" + startMinute);
```

```
Starts at 10:15
Starts at 10:15
```

First representation (lines 1–2): one `int`, `startMinutes`, holds the number of minutes since midnight. Ten hours is $10 \times 60 = 600$ minutes, plus 15, so the value is 615. To display it, line 2 uses `int` division and remainder from Topic 1.3:

Expression	Value	Meaning
<code>startMinutes / 60</code>	10	Whole hours in 615 minutes
<code>startMinutes % 60</code>	15	Minutes left over after those hours

Second representation (lines 4–6): two `int` variables, `startHour` and `startMinute`, hold 10 and 15 directly.

The concrete details are completely different: one number versus two, 615 versus 10 and 15. But the abstract property is identical: the class starts at 10:15. That gap between what the data means and how it is stored is exactly what data abstraction is about.

Now vary the situation. Suppose the schedule program needs to know how long a class lasts. With the first representation, a 10:15 start (615) and an 11:05 end (665) give $665 - 615 = 50$ minutes with one subtraction. With the second, you have to subtract hours and minutes separately and handle borrowing when the end minute is smaller than the start minute. Programmers choose a representation based on what the program needs to do with the data. But someone who only needs to know "the start time" does not care which one was chosen, and with good data abstraction, they should not have to.

WATCH OUT

A name that hides what the data means. A teammate finds this code and wants the end time to be one hour after the start:

```
1 int start = 615;
2 int later = start + 1;
3 System.out.println("Ends at " + later / 60 + ":" + later % 60);
```

```
Ends at 10:16
```

The name `start` says nothing about how the time is stored, so the teammate guessed that 615 counts hours and added 1. But the value counts minutes, so adding 1 moved the time one minute, not one hour. This is a logic error: the code compiles and runs, but the answer is wrong.

A good name carries the meaning the reader needs. With the name `startMinutes`, the unit is part of the abstraction, and adding an hour clearly means adding 60:

```
1 int startMinutes = 615;
2 int endMinutes = startMinutes + 60;
3 System.out.println("Ends at " + endMinutes / 60 + ":" + endMinutes % 60);
```

Ends at 11:15

PAUSE AND TRY

Question 3 Tracing

A store program represents a price as a whole number of cents. What does this code print, and what abstract property does the variable represent?

```
1 int priceInCents = 1250;
2 System.out.println("$" + priceInCents / 100 + "." + priceInCents % 100);
```

Question 4 Multiple choice

Which statement best describes data abstraction?

- A. Storing every piece of data as a `String` so that it can hold anything
- B. Giving data a name and meaning so that you can use it without referring to the details of how it is stored
- C. Writing data values directly into expressions, such as `price * 0.08`, instead of using variables
- D. Making sure each piece of data can be represented in only one possible way

3.1.3 Attributes in a Class Design

In Topic 1.12 you learned that a class is a blueprint that defines the **attributes** (what an object knows) and **behaviors** (what an object can do) of a kind of object. You used classes other people wrote, like `String` and `Scanner`. Starting in this unit, you will design and write your own. That raises a new question: when you plan a class, where does its data go, and how do you write the plan down?

Think about the variables you have written so far. Every one of them was declared inside a method, usually `main`. A variable declared inside a method belongs to that method: it is created when that line runs and is gone when the method finishes. That is fine for a temporary value like a loop counter. But a song in a music app needs to know its title the whole time the song object exists, no matter which behavior is running.

That is what an attribute is in Java. An **attribute** is a data abstraction defined in a class **outside any method or constructor**. Because it sits in the class itself rather than inside one method, it describes the object as a whole, and every behavior of the class can use it. (You will learn the Java syntax for writing attributes in Topic 3.3; for now, the focus is planning them.)

A class brings related data together under one name, which is data abstraction at the size of a whole class. Instead of juggling four separate variables called `title`, `artist`, `lengthInSeconds`, and `timesPlayed`, a program can work with one `Song` object that has all four.

Drawing a class diagram

Programmers plan classes with a **class diagram**: a box with three parts. The top part names the class, the middle part lists the attributes with their types, and the bottom part lists the behaviors as method headers. Here is a plan for a `Song` class in a music app:

```
+-----+
|           Song           |
+-----+
| Attributes                |
|   String title            |
|   String artist          |
|   int  lengthInSeconds   |
|   int  timesPlayed       |
+-----+
| Behaviors                 |
|   void play()             |
|   String getTitle()       |
|   int  getLengthInSeconds() |
+-----+
```

Read the attribute line `int lengthInSeconds` the same way you read a variable declaration: the type, then the name. Read the behavior lines the same way you read method headers in Topic 1.9: return type, name, and parameters. You may see other diagram styles that put the type after the name, like `lengthInSeconds : int`; they mean the same thing.

KEY IDEA

An attribute is data a class keeps outside any method or constructor, so it describes the whole object and every behavior can use it. A class diagram lists the class name, its attributes with types, and its behaviors as method headers.

WORKED EXAMPLE From a description to attributes

Here is a description of what a movie theater needs:

The theater sells tickets for showings of a movie. Each ticket is for one movie title and one seat, and has a price. The theater wants to know whether each ticket has been scanned at the door.

To find the attributes, look for the facts the program must remember about each ticket. Nouns and "whether" phrases are good clues. Then pick a type for each one using what you know from Unit 1:

Fact from the description	Attribute	Type	Why this type
one movie title	<code>movieTitle</code>	<code>String</code>	A title is text
one seat	<code>seat</code>	<code>String</code>	Seats like "F12" mix letters and digits
has a price	<code>price</code>	<code>double</code>	Prices can have cents
whether it has been scanned	<code>scanned</code>	<code>boolean</code>	It is either true or false

Notice what is **not** an attribute: "the theater sells tickets" describes an action, which makes it a behavior, not data. And the theater's street address is not mentioned, because a ticket does not need it to do its job. That is abstraction from 3.1.1 at work.

Now vary it: the theater adds, "Each ticket also remembers how many times someone tried to scan it." That one sentence adds one attribute, `int scanAttempts`, because it is a new fact the ticket must remember. When the requirements change, the attribute list changes with them.

WATCH OUT

Listing a temporary value as an attribute. A student planning a `Song` class adds `int i` to the attributes "because the play behavior uses a loop counter." A loop counter is only needed while one behavior is running; nothing about the song itself depends on it. Making it an attribute clutters the design with something that does not describe the object.

Ask of each candidate: "Does every object need to remember this, the whole time it exists?" A song's title, yes. A counter used for a few lines inside one method, no; that belongs inside the method.

PAUSE AND TRY

Question 5 Multiple choice

A programmer is designing a `GymMember` class. Which of these belongs in the class's list of attributes?

- A. `checkIn()`, because members check in at the front desk
- B. `int visitsThisMonth`, because each member must keep track of it the whole time
- C. `int k`, a counter used inside one method to print a list of classes
- D. The gym's parking-lot paint color, because the gym has a parking lot

Question 6 Short answer

A podcast app needs an `Episode` class. Here is the description: "Each episode has a title, a length in minutes, and a number that tells its position in the series. The app shows whether the listener has finished it." List the attributes with a type for each, and name one fact about episodes that you would leave out and why.

3.1.4 Instance Variables and Class Variables

In Topic 1.12 you saw that two `String` objects have the same kind of attribute (characters) but their own values: `"Ada"` and `"Grace Hopper"` hold different characters. Most attributes work that way.

But think about the movie theater again. Every ticket has its own seat, but all the tickets are for showings at the same theater, at the same address. Should each ticket store its own copy of the theater's name? If the theater were renamed, someone would have to update every ticket ever sold.

Java gives you two kinds of attribute for these two situations:

- An **instance variable** is an attribute whose value belongs to one instance. Every object of the class has its own copy, and changing one object's value does not affect any other object. A ticket's seat is an instance variable.
- A **class variable** is an attribute that is shared by all instances of the class. There is only one copy, and every object sees the same value. If it changes, every object sees the new value. The theater name is a class variable.

KEY IDEA

An instance variable has a separate value in each object. A class variable has one value shared by every object of the class.

WORKED EXAMPLE Three tickets, one theater

Suppose three tickets have been sold. Picture what each one holds:

```

ticket A      ticket B      ticket C
seat: "B12"   seat: "F3"     seat: "A1"
price: 12.5   price: 9.0     price: 15.0
  |           |             |
+-----+-----+
                |
        theaterName: "Riverside Cinema"
                (one value, shared by all)

```

Each ticket has its own `seat` and `price`; those are instance variables. There is only one `theaterName`, and all three tickets use it; that is a class variable.

Now trace two changes:

Change	Ticket A sees	Ticket B sees	Ticket C sees
Ticket B's seat changes to "F4"	seat "B12"	seat "F4"	seat "A1"
The theater is renamed "Riverside 8"	theater "Riverside 8"	theater "Riverside 8"	theater "Riverside 8"

Changing ticket B's seat touched only ticket B, because each ticket has its own seat. Renaming the theater changed what **all three** see, because there is only one theater name to change.

Now vary the class. A bank has accounts. Each account has its own `balance` (instance variable), but the bank pays the same `interestRate` on every account (class variable). If the bank raises its rate from 2% to 3%, every account immediately uses 3%, with one change. If the rate were an instance variable, the bank would have to update thousands of accounts one at a time, and missing even one would make that customer's interest wrong.

In a class diagram, mark a class variable so that readers can tell it apart. This guide labels it in the attribute list:

```
+-----+
|           MovieTicket           |
+-----+
| Attributes                       |
|   String seat                   |
|   double price                  |
|   String theaterName      (class variable) |
+-----+
```

(In Java, the keyword that makes an attribute a class variable is taught in Topic 3.7. The design decision comes first.)

WATCH OUT

Making per-object data shared. A student designs a `BankAccount` class and marks `balance` as a class variable "because every account has a balance." Then a customer deposits \$100 into account A, and every account in the bank suddenly shows the same new balance, because there is only one shared balance.

The test is not "does every object have this kind of data?" (every account has a balance, and every account uses an interest rate). The test is "should every object have the **same value**?" If each object needs its own value, it is an instance variable. If one value must be shared by all, it is a class variable.

PAUSE AND TRY

Question 7 Multiple choice

A `Student` class has the attributes `name`, `gradeLevel`, and `schoolName`. All the students in this program attend the same school. Which choice is the best design?

- A. All three are class variables, because every student has a name, a grade level, and a school
- B. `name` and `gradeLevel` are instance variables; `schoolName` is a class variable
- C. `schoolName` is an instance variable; `name` and `gradeLevel` are class variables
- D. All three are instance variables, because class variables can never be useful

Question 8 Tracing

A game has a `Player` class with an instance variable `score` and a class variable `level`. Player Kai has a score of 40, player Rosa has a score of 75, and `level` is 1. Then two things happen, in order: (1) Kai's score increases by 10. (2) The game moves everyone to level 2. Give Kai's and Rosa's score and level after each change.

3.1.5 Procedural Abstraction and Method Decomposition

So far this chapter has been about data. Classes also have behaviors, and behaviors can get complicated. Picture a single `main` method that plans a whole party: it counts guests, works out food, rounds up to whole pizzas, figures out cost, and prints a plan. Forty lines in one block is hard to read and hard to fix, because every detail of every step is mixed together.

You already know the tool that helps. In Topic 1.9 you learned **procedural abstraction**: you can use a method knowing only *what* it does, not *how* it does it, the way you use `Math.sqrt` without knowing how it computes square roots. Procedural abstraction also **gives a name to a process**. The name `sqrt` stands for a whole calculation, so a reader thinks "square root" instead of following every step.

You can use that idea in your own programs. **Method decomposition** means breaking a large behavior into smaller behaviors and writing a separate method for each one. The big behavior then becomes a short list of calls, and each call's name says what that step does. It works like a recipe that says "make the sauce" and puts the sauce instructions on a separate card.

KEY IDEA

Procedural abstraction gives a process a name so it can be used by knowing what it does, not how. Method decomposition breaks a large behavior into smaller methods, one for each smaller behavior.

WORKED EXAMPLE Decomposing a pizza-party plan

Here is a party planner written as one block. Read it and try to say what each line is for:

PizzaPartyOneBlock.java

```
1 public class PizzaPartyOneBlock
2 {
3     public static void main(String[] args)
4     {
5         int guests = 13;
6         int slices = guests * 3;
7         int pizzas = slices / 8;
8         if (slices % 8 != 0)
9         {
10             pizzas++;
11         }
12         System.out.println("Guests: " + guests);
13         System.out.println("Pizzas to order: " + pizzas);
14     }
15 }
```

```
Guests: 13
Pizzas to order: 5
```

It works, but the reader has to work out for themselves that lines 7–11 round up to whole pizzas. Here is the same plan decomposed into methods. (As in Topic 1.9, read past the words `public static` in each header; they are explained in later topics.)

PizzaParty.java

```

1  public class PizzaParty
2  {
3      public static void main(String[] args)
4      {
5          int guests = 13;
6          int slices = slicesNeeded(guests);
7          int pizzas = pizzasNeeded(slices);
8          printPlan(guests, pizzas);
9      }
10
11     public static int slicesNeeded(int people)
12     {
13         return people * 3;
14     }
15
16     public static int pizzasNeeded(int slices)
17     {
18         int pizzas = slices / 8;
19         if (slices % 8 != 0)
20         {
21             pizzas++;
22         }
23         return pizzas;
24     }
25
26     public static void printPlan(int people, int pizzas)
27     {
28         System.out.println("Guests: " + people);
29         System.out.println("Pizzas to order: " + pizzas);
30     }
31 }

```

```

Guests: 13
Pizzas to order: 5

```

Look at `main` (lines 3–9) first. It is now four lines, and each one reads like a step of the plan: get the guest count, find the slices needed, find the pizzas needed, print the plan. Each smaller behavior has its own method:

Method	What it does (all a caller needs)	How it does it (hidden from the caller)
<code>slicesNeeded</code>	Returns the slices needed for some number of people	Multiplies by 3 slices each
<code>pizzasNeeded</code>	Returns the whole pizzas needed for some slices	Divides by 8 and adds 1 if any slices are left over
<code>printPlan</code>	Prints the guest count and pizza count	Two <code>println</code> statements

Trace the calls: `slicesNeeded(13)` returns $13 \times 3 = 39$. `pizzasNeeded(39)` computes $39 / 8 = 4$ with int division; $39 \% 8$ is 7, not 0, so `pizzas` goes up to 5 and 5 is returned. Then `printPlan(13, 5)` prints the two lines. The output is exactly the same as the one-block version. Decomposition changed how the program is organized, not what it does.

Decomposition is also a planning tool, before any code exists. Suppose a `LibraryBook` class needs a behavior "check out the book." Broken into smaller behaviors, it might be:

1. Check whether the book is available.
2. Record who is borrowing it.
3. Set the due date to two weeks from today.

Each step can become its own method in the plan, such as `boolean isAvailable()`, `void recordBorrower(String name)`, and `void setDueDate()`. The big behavior `checkout` then just uses those three. Each small method is easier to write and test on its own, and a method like `isAvailable` can be reused by other behaviors, such as a search page that shows which books are on the shelf.

WATCH OUT

A method name that describes the how instead of the what. A student names the rounding method `divideAndMaybeAddOne`. Now every reader of `main` sees the steps of the calculation again, and if the calculation ever changes (as it will in 3.1.7), the name becomes wrong.

A procedural abstraction's name should say **what** the method does, so callers never need to think about how: `pizzasNeeded`, not `divideAndMaybeAddOne`.

PAUSE AND TRY

Question 9 Tracing

Trace this program and give its exact output.

CarWash.java

```
1 public class CarWash
2 {
3     public static void main(String[] args)
4     {
5         printSign();
6         int total = washCost(2) + waxCost(2);
7         System.out.println("Total: $" + total);
8     }
9
10    public static void printSign()
11    {
12        System.out.println("Sparkle Car Wash");
13    }
14
15    public static int washCost(int cars)
16    {
17        return cars * 8;
18    }
19
20    public static int waxCost(int cars)
21    {
22        System.out.println("Adding wax");
23        return cars * 5;
24    }
25 }
```

Question 10 Multiple choice

In `PizzaParty.java`, a teammate only reads `main`. What do they need to know about `pizzasNeeded` in order to use it correctly?

- A. That it divides by 8 and uses the remainder operator
- B. That it takes a number of slices and returns how many whole pizzas are needed
- C. Every line inside it, so they can check that it is correct each time they call it
- D. Nothing at all, not even what it returns

3.1.6 Generalizing with Parameters

Decomposition breaks one big behavior into pieces. A different problem shows up when a program does almost the same thing in several places. Look at this program, which prints three rows of stars of different lengths:

StarsRepeated.java

```
1 public class StarsRepeated
2 {
3     public static void main(String[] args)
4     {
5         for (int i = 0; i < 5; i++)
6         {
7             System.out.print("*");
8         }
9         System.out.println();
10        for (int i = 0; i < 3; i++)
11        {
12            System.out.print("*");
13        }
14        System.out.println();
15        for (int i = 0; i < 8; i++)
16        {
17            System.out.print("*");
18        }
19        System.out.println();
20    }
21 }
```

```
*****
***
*****
```

The three loops with their `println` calls (lines 5–9, 10–14, and 15–19) are identical except for one number: 5, 3, or 8. That is **duplicated code**: the same steps written out more than once. It makes the program longer, and every copy is another place for a mistake.

The fix is to notice the **shared features** (everything that is the same in each copy), put them into one method, and turn the one thing that differs into a parameter. Recall from Topic 1.9 that a parameter is a variable in the method header that receives a copy of the argument each time the method is called. Because the argument can be different on every call, a parameter makes a method **general**: it works for a whole range of input values, not just one. Writing the work once and calling it many times is **code reuse**, and it helps manage complexity.

KEY IDEA

A procedural abstraction can pull out the shared features of duplicated code into one method. Parameters generalize that method so it can be reused with many different arguments.

WORKED EXAMPLE One method replaces three loops

Here is the star program rewritten with one general method:

StarsMethod.java

```

1  public class StarsMethod
2  {
3      public static void main(String[] args)
4      {
5          printStars(5);
6          printStars(3);
7          printStars(8);
8      }
9
10     public static void printStars(int count)
11     {
12         for (int i = 0; i < count; i++)
13         {
14             System.out.print("*");
15         }
16         System.out.println();
17     }
18 }

```

```

*****
***
*****

```

The loop now appears once, inside `printStars` (lines 10–17). The number that used to change from copy to copy is now the parameter `count`, and the loop condition on line 12 uses `count` instead of a fixed number. Trace the three calls:

Call	count receives	Loop runs for i =	Printed
<code>printStars(5)</code>	5	0, 1, 2, 3, 4	*****
<code>printStars(3)</code>	3	0, 1, 2	***
<code>printStars(8)</code>	8	0 through 7	*****

Same output as before, with one loop instead of three. A fourth row of 12 stars now costs one line, `printStars(12);`, instead of five more lines of loop.

Now vary what is generalized. Here is a method with no parameter:

WelcomeFixed.java

```

1  public class WelcomeFixed
2  {
3      public static void main(String[] args)
4      {
5          printWelcomeAda();
6      }
7
8      public static void printWelcomeAda()
9      {
10         System.out.println("Welcome back, Ada!");
11     }
12 }

```

```
Welcome back, Ada!
```

It can greet only Ada. To greet Grace, you would need a second method, `printWelcomeGrace`, with the same body except for the name. Instead, make the name a parameter:

`WelcomeAnyone.java`

```
1 public class WelcomeAnyone
2 {
3     public static void main(String[] args)
4     {
5         printWelcome("Ada");
6         printWelcome("Grace");
7         String nextUser = "Alan";
8         printWelcome(nextUser);
9     }
10
11     public static void printWelcome(String name)
12     {
13         System.out.println("Welcome back, " + name + "!");
14     }
15 }
```

```
Welcome back, Ada!
Welcome back, Grace!
Welcome back, Alan!
```

The shared feature is the greeting pattern, "Welcome back, ...!", and the part that varies, the name, became the parameter `name`. The argument can be a literal like `"Ada"` or a variable like `nextUser`; either way, its value is copied into `name`.

WATCH OUT

Fixing one copy of duplicated code but not the other. A game gives a bonus at the end of each level. The designer changes the bonus from +10 to +20, but the bonus formula was copied in two places, and only one was updated:

`BonusCopies.java`

```
1 public class BonusCopies
2 {
3     public static void main(String[] args)
4     {
5         int levelOneScore = 40;
6         int levelOneTotal = levelOneScore * 2 + 20;
7         System.out.println("Level 1 total: " + levelOneTotal);
8
9         int levelTwoScore = 40;
10        int levelTwoTotal = levelTwoScore * 2 + 10;
11        System.out.println("Level 2 total: " + levelTwoTotal);
12    }
13 }
```

```
Level 1 total: 100  
Level 2 total: 90
```

Both levels had the same score of 40, yet the players get different totals. Nothing crashes, so this logic error is easy to miss. It happened because the same rule lived in two places. With one method, the rule lives in exactly one place, and changing it changes every use:

BonusMethod.java

```
1 public class BonusMethod  
2 {  
3     public static void main(String[] args)  
4     {  
5         int levelOneScore = 40;  
6         System.out.println("Level 1 total: " + withBonus(levelOneScore));  
7  
8         int levelTwoScore = 40;  
9         System.out.println("Level 2 total: " + withBonus(levelTwoScore));  
10    }  
11  
12    public static int withBonus(int score)  
13    {  
14        return score * 2 + 20;  
15    }  
16 }
```

```
Level 1 total: 100  
Level 2 total: 100
```

PAUSE AND TRY

Question 11 Debugging

A student replaced three star loops with a general method, but this program prints the wrong rows. It should print rows of 5, 3, and 8 stars. Find the error and write the corrected line.

StarsIgnoreParam.java

```
1 public class StarsIgnoreParam
2 {
3     public static void main(String[] args)
4     {
5         printStars(5);
6         printStars(3);
7         printStars(8);
8     }
9
10    public static void printStars(int count)
11    {
12        for (int i = 0; i < 5; i++)
13        {
14            System.out.print("*");
15        }
16        System.out.println();
17    }
18 }
```

Question 12 Multiple choice

A program contains two methods: `printMathGrade()`, which prints Math: A, and `printArtGrade()`, which prints Art: B. Their bodies are identical except for the subject and the letter. Which single method could replace both, so that the old calls become `printGrade("Math", "A");` and `printGrade("Art", "B");`?

- A. `public static void printGrade(String subject, String letter)`
- B. `public static void printGrade()`
- C. `public static void printGrade(String subject)`
- D. `public static void printMathAndArtGrades(String letter)`

3.1.7 Changing How a Method Works

Programs keep changing after they are written. A method might be too slow, or someone might find a shorter way to compute the same thing. But a method like `pizzasNeeded` may be called from dozens of places, written by other programmers. Does improving it mean finding and warning every one of them?

Procedural abstraction answers this. Callers depend only on two things: the method's **signature** (its name and parameter types, from Topic 1.9) and **what it does** (the result it promises). They never depended on how it works inside. So a programmer may change the inside of a method, to make it faster, more efficient, or use less memory, without telling the method's users, **as long as the signature and what the method does stay the same**.

KEY IDEA

You can change how a method works without telling the programmers who call it, as long as its signature and what it does stay the same.

WORKED EXAMPLE Rewriting pizzasNeeded

Here is `pizzasNeeded` from 3.1.5 with a `main` that tests it on three inputs:

`PizzaIfVersion.java`

```
1 public class PizzaIfVersion
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(pizzasNeeded(39));
6         System.out.println(pizzasNeeded(40));
7         System.out.println(pizzasNeeded(41));
8     }
9
10    public static int pizzasNeeded(int slices)
11    {
12        int pizzas = slices / 8;
13        if (slices % 8 != 0)
14        {
15            pizzas++;
16        }
17        return pizzas;
18    }
19 }
```

```
5
5
6
```

A teammate finds a one-line way to round up: add 7 before dividing by 8. Here is the program with only the method's body changed (lines 10–13); `main` is untouched:

`PizzaFormulaVersion.java`

```
1 public class PizzaFormulaVersion
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(pizzasNeeded(39));
6         System.out.println(pizzasNeeded(40));
7         System.out.println(pizzasNeeded(41));
8     }
9
10    public static int pizzasNeeded(int slices)
11    {
12        return (slices + 7) / 8;
13    }
14 }
```

```
5
5
6
```

Check that the new body does the same thing for each test:

slices	Old: slices / 8, then +1 if leftover	New: (slices + 7) / 8	Same?
39	4, leftover 7, so 5	$46 / 8 = 5$	Yes
40	5, no leftover, so 5	$47 / 8 = 5$	Yes
41	5, leftover 1, so 6	$48 / 8 = 6$	Yes

Adding 7 lifts any number with a leftover up to at least the next multiple of 8, so the division counts one more pizza; a number that is already a multiple of 8 stays short of the next multiple, so nothing extra is counted. The signature is still `pizzasNeeded(int)`, and it still returns the number of whole pizzas, so every caller keeps working with no changes and no warning needed.

Now vary the kind of change. Two other changes look small but break the promise.

Changing the signature. Suppose the teammate makes the method more general by adding a parameter for slices per pizza, but the callers still pass one argument:

PizzaNewSignature.java

```

1 public class PizzaNewSignature
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(pizzasNeeded(39));
6     }
7
8     public static int pizzasNeeded(int slices, int slicesPerPizza)
9     {
10        return (slices + slicesPerPizza - 1) / slicesPerPizza;
11    }
12 }
```

PizzaNewSignature.java:5: error: method pizzasNeeded in class PizzaNewSignature cannot be applied to given types;

```

    System.out.println(pizzasNeeded(39));
                        ^
```

required: int,int

found: int

(The exact wording can vary between Java versions.) The signature is now `pizzasNeeded(int, int)`, so every existing call with one argument fails to compile. This is a change callers must be told about, because they have to change their code.

Changing what it does. Suppose the teammate "simplifies" the body to just divide:

PizzaForgotRoundUp.java

```
1 public class PizzaForgotRoundUp
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(pizzasNeeded(39));
6         System.out.println(pizzasNeeded(40));
7         System.out.println(pizzasNeeded(41));
8     }
9
10    public static int pizzasNeeded(int slices)
11    {
12        return slices / 8;
13    }
14 }
```

4
5
5

The signature is the same, so everything compiles. But the method no longer returns the number of whole pizzas needed: for 39 slices it says 4 pizzas, which is only 32 slices. Every caller now gets wrong answers without any error message. That is worse than a compile error, because nobody is warned.

WATCH OUT

"If it still compiles, the change is safe." The signature is only half of the promise.

`PizzaForgotRoundUp.java` compiles perfectly and is still broken, because it changed what the method does. Before changing a method's inside, test the new version on the same inputs as the old one, including edge cases like an exact multiple of 8, and confirm that every result matches.

PAUSE AND TRY**Question 13** Multiple choice

Many programs call `public static int pizzasNeeded(int slices)`. Which change can be made **without** telling the programmers who call it?

- A. Renaming the method to `wholePizzas`
- B. Changing the parameter type to `double`, so the header is `pizzasNeeded(double slices)`
- C. Rewriting the body as `return (slices + 7) / 8;`
- D. Rewriting the body as `return slices / 8;` because it is shorter

Question 14 Short answer

Your program uses the `String` method `indexOf` from Topic 1.15. Suppose the people who build Java rewrite the inside of `indexOf` in a new version to make it faster. Explain whether your program needs to change, and what must be true for your answer to hold.

3.1.8 Designing a Class Before Coding

When you write a short program, you might start typing right away. A class is different. Its attributes are used by all its behaviors, and other programmers will call its methods. If you discover halfway through that a behavior needs data you never planned, or that an attribute should have been shared, you may have to rewrite much of what you wrote.

That is why it helps to **design** each class before implementing it: decide its attributes and its behaviors first, then write the code. You have now seen every piece of a design:

- **Abstraction** (3.1.1) decides which details belong at all.
- **Attributes** (3.1.3), marked as **instance** or **class** variables (3.1.4), hold the data.
- **Behaviors** (3.1.5), named for what they do, decomposed into smaller methods and generalized with parameters (3.1.6).

A design can be written in **natural language** (plain sentences or a list) or drawn as a **class diagram**. Both record the same decisions; a diagram is quicker to scan, and sentences are easier to share with someone who does not know the notation.

KEY IDEA

Before writing a class, design it: choose its attributes and behaviors. Record the design in natural language or in a class diagram.

WORKED EXAMPLE Designing a WaterBottle tracker

A fitness app's description says:

The app tracks how much water a user drinks from their bottle. Each bottle has a capacity in milliliters and a current amount. The user can drink some amount, and refill the bottle to capacity. The app shows how full the bottle is as a percent. Every bottle uses the same daily goal of 2000 mL.

Step 1: Find the attributes. The facts each bottle must remember are its capacity and its current amount: different for each bottle, so instance variables. The daily goal is the same for every bottle, so it is a class variable. The bottle's color is not mentioned and not needed, so abstraction leaves it out.

Step 2: Find the behaviors. The verbs give them: drink some amount, refill, and show how full it is. "Drink some amount" needs to know how much, so it gets a parameter. "How full" sends back a number, so it returns a value.

Step 3: Write the design in natural language.

- A `WaterBottle` has a capacity (whole milliliters) and a current amount (whole milliliters), its own for each bottle. All bottles share one daily goal (whole milliliters).
- A bottle can drink a given amount, refill to capacity, and report how full it is as a percent.

Step 4: Draw it as a class diagram.

```

+-----+
|           WaterBottle           |
+-----+
| Attributes                       |
|   int capacity                   |
|   int currentAmount              |
|   int dailyGoal                  (class variable) |
+-----+
| Behaviors                       |
|   void drink(int amount)         |
|   void refill()                  |
|   double percentFull()           |
+-----+

```

Check every behavior against the attributes: `drink` lowers `currentAmount`; `refill` sets `currentAmount` to `capacity`; `percentFull` compares `currentAmount` with `capacity`. Every behavior has the data it needs, so the design is complete for this description.

Now vary the requirements. The app adds: "The bottle remembers how many times it has been refilled." That adds one instance variable, `int refillCount`, and the `refill` behavior now also updates it. Changing a design on paper takes a minute; changing a class after hundreds of lines use it takes much longer. That is the payoff of designing first.

WATCH OUT

A behavior with no data to work with. A student's `LibraryBook` design has the attributes `title` and `author` and the behavior `boolean isOverdue()`. It looks fine until you ask how `isOverdue` could possibly decide. Overdue means "past the due date," and the design never stores a due date. The class could not be written as planned.

Fix it by checking each behavior: "What data does this need?" Here, add a due date attribute (for example `int dueDay`, a day number in the school year). Catching this in the design costs one line; catching it while coding can mean reworking several methods.

PAUSE AND TRY

Question 15 Debugging

This class diagram is meant to match the description: "Each parking spot has a spot number and knows whether it is taken. Every spot in the garage charges the same hourly rate. A spot can be marked taken or free, and can report the cost of parking for a given number of hours." Find **two** errors in the diagram and describe how to fix each one.

```

+-----+
|           ParkingSpot           |
+-----+
| Attributes                       |
|   int spotNumber      (class variable) |
|   boolean taken      |
|   double hourlyRate  |
+-----+
| Behaviors                 |
|   void markTaken()      |
|   void markFree()      |
|   double cost()         |
+-----+

```

Putting it together

Here is a short program that uses the chapter's ideas together: a reading planner that tells a student how many days it will take to finish a book.

ReadingPlan.java

```

1  public class ReadingPlan
2  {
3      public static void main(String[] args)
4      {
5          int days = daysToFinish(250, 30);
6          printPlan("Hatchet", days);
7          days = daysToFinish(90, 30);
8          printPlan("Frindle", days);
9      }
10
11     public static int daysToFinish(int pages, int pagesPerDay)
12     {
13         int days = pages / pagesPerDay;
14         if (pages % pagesPerDay != 0)
15         {
16             days++;
17         }
18         return days;
19     }
20
21     public static void printPlan(String title, int days)
22     {
23         System.out.println(title + ": " + days + " days");
24     }
25 }

```

```

Hatchet: 9 days
Frindle: 3 days

```

Look at it through each idea from the chapter:

- **Data abstraction (3.1.2).** The names `pages`, `pagesPerDay`, `days`, and `title` say what each value means. Nobody reading `main` has to think about how they are stored.
- **Method decomposition (3.1.5).** The job "plan the reading" is split into two smaller behaviors: compute the days (`daysToFinish`) and print the result (`printPlan`). `main` reads as a list of steps.
- **Procedural abstraction (3.1.5).** A reader of `main` needs only to know that `daysToFinish` returns the whole number of days needed. How it rounds up (lines 13–17) is hidden.
- **Generalizing with parameters (3.1.6).** One `daysToFinish` method works for any book and any reading speed; one `printPlan` method prints any title. Without parameters, each book would need its own copy of the calculation.

Trace the two calls to `daysToFinish`:

Call	pages / pagesPerDay	pages % pagesPerDay	Leftover?	Returns
<code>daysToFinish(250, 30)</code>	8	10	yes, +1	9
<code>daysToFinish(90, 30)</code>	3	0	no	3

For Hatcher, 8 full days covers 240 pages, and the last 10 pages need a ninth day. For Frindle, 3 days covers exactly 90 pages, so no extra day is added.

If this planner became a class, the design step (3.1.8) would start by asking what each book remembers: its title and page count, as instance variables. Writing classes like that in Java begins in Topic 3.3.

Vocabulary

Term	Definition	Example
abstraction	Reducing complexity by hiding details that do not matter for the problem and keeping those that do	A library program keeps a book's title but not its weight
data abstraction	Separating what data means from how it is stored, by working with it through a name	<code>startMinutes</code> means "the start time"
representation	The specific way a piece of data is stored	10:15 stored as the <code>int</code> 615
attribute	A data abstraction defined in a class, outside any method or constructor	A song's title
class diagram	A box listing a class's name, its attributes with types, and its behaviors as method headers	The <code>WaterBottle</code> diagram in 3.1.8
instance variable	An attribute with a separate value in each object of the class	Each ticket's <code>seat</code>
class variable	An attribute with one value shared by all objects of the class	The theater's name, shared by all tickets
procedural abstraction	Giving a process a name so it can be used by knowing what it does, not how it does it	Calling <code>pizzasNeeded(39)</code> without reading its body

method decomposition	Breaking a large behavior into smaller behaviors, each written as its own method	Party planning split into <code>slicesNeeded</code> and <code>pizzasNeeded</code>
duplicated code	The same steps written out in more than one place	Three nearly identical star loops
code reuse	Writing code once and using it many times, for example by calling one method from several places	<code>printStars(5)</code> , <code>printStars(3)</code> , <code>printStars(8)</code>
design (of a class)	A plan of a class's attributes and behaviors, made before writing its code	The <code>WaterBottle</code> sentences and diagram

Chapter summary

- **Abstraction** reduces complexity by hiding details that do not matter for the problem and keeping the ones that do; which details matter depends on what the program is for.
- **Data abstraction** separates what data means from how it is represented. A good name lets you use data without knowing its storage details, whether the data is one variable, a class, or a set of values.
- A class's **attributes** are kept outside any method or constructor. An **instance variable** has its own value in each object; a **class variable** has one value shared by every object.
- **Procedural abstraction** names a process so it can be used by knowing what it does, not how. **Method decomposition** splits large behaviors into smaller methods, and **parameters** let one general method replace duplicated code and be reused with many arguments.
- The inside of a method can change (to be faster or smaller) without telling its callers, as long as its signature and what it does stay the same.
- Before coding a class, **design** it: choose its attributes and behaviors and record them in natural language or a **class diagram**.

Review questions

Question 16 Multiple choice

A navigation app shows drivers the fastest route. Which choice uses **both** data abstraction and procedural abstraction?

- The app stores each road's speed limit and length under names like `speedLimit` and `lengthInMiles`, and calls a method `fastestRoute(start, end)` without the driver code knowing how the route is found.
- The app stores every photo ever taken of each road, so that no detail is lost.
- The app copies the route-finding steps into every screen that needs a route.
- The app stores road lengths as unnamed numbers written directly into each calculation.

Question 17 Debugging

The program below is supposed to print the days needed to finish each book, rounding up any partial day. Wonder has 310 pages, so at 30 pages a day it should take 11 days, but the program prints `Wonder: 10 days`. Find the error, explain why it happened, and write the corrected line.

ReadingPlanCopy.java

```
1 public class ReadingPlanCopy
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Hatchet: " + daysToFinish(250, 30) + " days");
6         int wonderDays = 310 / 30;
7         System.out.println("Wonder: " + wonderDays + " days");
8     }
9
10    public static int daysToFinish(int pages, int pagesPerDay)
11    {
12        int days = pages / pagesPerDay;
13        if (pages % pagesPerDay != 0)
14        {
15            days++;
16        }
17        return days;
18    }
19 }
```

Question 18 Multiple choice

Many programs call `public static int daysToFinish(int pages, int pagesPerDay)` from the Putting it together section. Which change could be made without telling any of those programmers?

- A. Swapping the parameters to `daysToFinish(int pagesPerDay, int pages)`
- B. Changing the body to `return (pages + pagesPerDay - 1) / pagesPerDay;`
- C. Changing the body to return the number of **hours** needed instead of days
- D. Adding a third parameter, `int startDay`

Question 19 Short answer

A school is designing a `Locker` class from this description: "Each locker has a locker number and a combination. A locker knows whether it is assigned to a student. All lockers in the building have the same height in inches. A locker can be assigned to a student by name, and can report whether it is assigned." (a) For each of the four attributes, say whether it should be an instance variable or a class variable, and why. (b) What parameter does the "assign" behavior need, and what type of value should the "report" behavior return?

Question 20 Tracing

Here are two versions of a method that returns $1 + 2 + \dots + n$. Each program is complete; only the method body differs.

SumToLoop.java

```
1 public class SumToLoop
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(sumTo(4));
6         System.out.println(sumTo(10));
7     }
8
9     public static int sumTo(int n)
10    {
11        int total = 0;
12        for (int k = 1; k <= n; k++)
13        {
14            total += k;
15        }
16        return total;
17    }
18 }
```

SumToFormula.java

```
1 public class SumToFormula
2 {
3     public static void main(String[] args)
4     {
5         System.out.println(sumTo(4));
6         System.out.println(sumTo(10));
7     }
8
9     public static int sumTo(int n)
10    {
11        return n * (n + 1) / 2;
12    }
13 }
```

(a) What does each program print? (b) For the call `sumTo(10)`, how many times does the statement `total += k;` run in the first version? (c) Can the second version replace the first without telling callers? Explain.

Question 21 Multiple choice

A music app's `Song` class has the attributes `title` (instance variable) and `playsToday` and the behavior `void play()`, which adds 1 to `playsToday`. The designer marked `playsToday` as a **class variable** by mistake. Song X and Song Y each start with 0 plays today. Song X is played twice and Song Y once. What will Song Y report for `playsToday`?

- A. 1
- B. 2
- C. 3
- D. 0

Reflection

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

- ☐ I can explain abstraction and choose which details a program should keep for a given problem. Reread: 3.1.1 Abstraction: Keeping What Matters
- ☐ I can explain how data abstraction separates what data means from how it is stored. Reread: 3.1.2 Data Abstraction: Naming Data
- ☐ I can explain what an attribute is and read or draw a class diagram. Reread: 3.1.3 Attributes in a Class Design
- ☐ I can decide whether an attribute should be an instance variable or a class variable. Reread: 3.1.4 Instance Variables and Class Variables
- ☐ I can explain procedural abstraction and break a large behavior into smaller methods. Reread: 3.1.5 Procedural Abstraction and Method Decomposition
- ☐ I can explain how a parameter lets one method replace duplicated code. Reread: 3.1.6 Generalizing with Parameters
- ☐ I can decide whether a change to a method needs to be announced to the programmers who call it. Reread: 3.1.7 Changing How a Method Works
- ☐ I can design a class by listing its attributes and behaviors in words or in a class diagram. Reread: 3.1.8 Designing a Class Before Coding

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. To predict when a bus reaches a stop, the program needs to know which route it is on, where it is now, and which stops come next. Those are the details that matter for this question.

- **A** is wrong: none of these details help predict an arrival time. They are exactly the kind of detail abstraction hides.
- **C** is wrong: keeping everything is the opposite of abstraction. It buries the few useful details among many useless ones.
- **D** is wrong: this hides too much. With only the route number, the program cannot tell where the bus is or when it will arrive.

Reread *3.1.1 Abstraction: Keeping What Matters* if you chose differently.

Question 2

Model answer: The nurse's program should keep a student's allergies, which the bus program does not need. The bus program should keep the student's home address or bus stop, which the nurse's program does not need. The same student gets a different set of details in each program because abstraction keeps the details that matter for the question the program answers, and the two programs answer different questions.

A complete answer must:

- give a sensible detail for each program (allergies, medications, or an emergency contact for the nurse; address, bus stop, or pickup time for the bus), and
- explain that which details to keep depends on the purpose of the program.

Reread *3.1.1 Abstraction: Keeping What Matters* if your answer differed.

Question 3

It prints `$12.50`.

Expression	Value
<code>priceInCents / 100</code>	12
<code>priceInCents % 100</code>	50

The string is built left to right: `"$"` + 12 gives `"$12"`, then `"."` gives `"$12."`, then 50 gives `"$12.50"`. The concrete representation is the `int` 1250 (cents), but the abstract property is a price of twelve dollars and fifty cents. A reader who sees the name `priceInCents` knows both what it means and what unit it uses.

Reread *3.1.2 Data Abstraction: Naming Data* if your answer differed.

Question 4

B. Data abstraction separates what data means from how it is represented, and it does this by giving the data a name you can think about without the storage details.

- **A** is wrong: the type chosen is part of the representation, and storing everything as text has nothing to do with separating meaning from storage.
- **C** is wrong: unnamed values are the opposite of data abstraction. A reader sees a bare `0.08` and must guess what it means.
- **D** is wrong: the worked example in 3.1.2 showed one start time represented two different ways. Data abstraction is what lets users ignore which way was chosen.

Reread 3.1.2 *Data Abstraction: Naming Data* if you chose differently.

Question 5

B. An attribute is data each member object keeps, defined outside any method. The number of visits is a fact about the member that several behaviors may need, so it is an attribute.

- **A** is wrong: `checkIn()` is an action, so it is a behavior, not an attribute.
- **C** is wrong: a counter used inside one method only matters while that method runs, so it belongs inside the method, not in the class.
- **D** is wrong: it has nothing to do with a member. Abstraction hides it.

Reread 3.1.3 *Attributes in a Class Design* if you chose differently.

Question 6

Model answer:

Attribute	Type
title	String
lengthInMinutes	int (or double if partial minutes matter)
episodeNumber	int
finished	boolean

I would leave out something like the microphone used to record it, because nothing the app does with episodes depends on it.

A complete answer must:

- list the four facts from the description as attributes with sensible types (text as `String`, counts as `int`, a yes/no fact as `boolean`), and
- leave out a detail with the reason that the program does not need it.

Reread 3.1.3 *Attributes in a Class Design* and 3.1.1 *Abstraction: Keeping What Matters* if your answer differed.

Question 7

B. Each student has their own name and grade level, so those need a separate value in each object. All students share one school, so one shared `schoolName` is enough, and renaming the school updates everyone at once.

- **A** is wrong: it uses the wrong test. Every student having a name does not mean every student has the *same* name. Making `name` shared would give every student the same name.
- **C** is wrong: it swaps the two kinds. Names differ from student to student, while the school is the same for everyone.
- **D** is wrong: making `schoolName` an instance variable would work, but it would store the same value many times and require changing every object if the school were renamed. A shared value is exactly what class variables are for.

Reread 3.1.4 *Instance Variables and Class Variables* if you chose differently.

Question 8

After change	Kai's score	Kai's level	Rosa's score	Rosa's level
Start	40	1	75	1
(1) Kai's score + 10	50	1	75	1
(2) Level becomes 2	50	2	75	2

Change (1) affects only Kai, because `score` is an instance variable and each player has their own. Change (2) affects both players, because `level` is a class variable with one shared value. If you changed Rosa's score in step (1), you treated `score` as shared.

Reread 3.1.4 *Instance Variables and Class Variables* if your answer differed.

Question 9

```
Sparkle Car Wash
Adding wax
Total: $26
```

Step	What runs	Printed / returned
1	<code>printSign()</code> (line 5)	prints Sparkle Car Wash
2	<code>washCost(2)</code> (line 6)	returns $2 \times 8 = 16$
3	<code>waxCost(2)</code> (line 6)	prints Adding wax, returns $2 \times 5 = 10$
4	<code>total = 16 + 10</code> (line 6)	total is 26
5	line 7	prints Total: \$26

Adding wax prints before the total because `waxCost` must run, and print its line, before line 6 can finish computing `total`. `main` is a short list of named steps; each method hides its own details.

Reread 3.1.5 *Procedural Abstraction and Method Decomposition* if your answer differed.

Question 10

B. Procedural abstraction means a caller needs to know what the method does (its parameter, its return value, and their meaning) but not how it does it.

- **A** is wrong: that is the *how*, which procedural abstraction hides from callers.
- **C** is wrong: rereading the body at every call would throw away the benefit of the abstraction. The method's name and description tell callers what they need.
- **D** is wrong: abstraction hides the *how*, not the *what*. Without knowing what it returns, the teammate could not use the result correctly.

Reread 3.1.5 *Procedural Abstraction and Method Decomposition* if you chose differently.

Question 11

The program prints `*****` three times. Line 12 still uses the fixed number 5 from one of the old copies, so the method ignores its parameter. The value passed in arrives in `count`, but the loop never looks at it.

Corrected line 12:

```
1 for (int i = 0; i < count; i++)
```

Now each call's argument controls the number of stars, and the program prints rows of 5, 3, and 8, as in `StarsMethod.java`. A parameter only generalizes a method if the body actually uses it in place of the value that used to vary.

Reread 3.1.6 *Generalizing with Parameters* if you did not find the fixed 5.

Question 12

A. The shared feature is the pattern "subject, colon, letter grade." The two things that differ between the old methods, the subject and the letter, both become parameters, so one method handles both calls and any other subject too.

- **B** is wrong: with no parameters, the method can only print one fixed line. It cannot accept "Math" and "A", and the calls would not compile.
- **C** is wrong: it generalizes the subject but not the letter, so the letter would still be fixed inside the body, and the two-argument calls would not compile.
- **D** is wrong: its name ties it to two specific subjects, and with one parameter it cannot receive both the subject and the letter.

Reread 3.1.6 *Generalizing with Parameters* if you chose differently.

Question 13

C. It keeps the same signature and returns the same result as the old body (the table in 3.1.7 checks it on both sides of a multiple of 8), so callers are not affected.

- **A** is wrong: the name is part of the signature. Every call to `pizzasNeeded` would fail to compile.
- **B** is wrong: the parameter types are part of the signature, so this is a different method. Callers would have to be told, and the body would need changes too.
- **D** is wrong: the signature stays the same, but the method stops rounding up, so it no longer does what it promised. 39 slices would give 4 pizzas.

Reread 3.1.7 *Changing How a Method Works* if you chose differently.

Question 14

Model answer: My program does not need to change. I use `indexOf` through procedural abstraction: I depend only on its signature and on what it does (return the index of the first match, or -1 if there is none), not on how it searches. As long as the new version keeps the same signature and still returns the same results, my calls work exactly as before, just faster.

A complete answer must:

- say the program does not need to change, and
- state the condition: the signature and what the method does (its results) must stay the same.

Reread 3.1.7 *Changing How a Method Works* if your answer differed.

Question 15

Error 1: `spotNumber` is marked as a class variable, but each spot has its own number. As a class variable, every spot would share one number. **Fix:** make `spotNumber` an instance variable (remove the label), and mark `hourlyRate` as the class variable, since every spot charges the same rate.

Error 2: `cost()` has no parameter, but the description says it reports the cost "for a given number of hours." The behavior needs the hours to compute the cost. **Fix:** change it to `double cost(int hours)`.

Corrected attribute and behavior lines:

```
|  int spotNumber          |
|  boolean taken           |
|  double hourlyRate      (class variable) |
|  ...                     |
|  double cost(int hours)  |
```

Reread 3.1.8 *Designing a Class Before Coding* and 3.1.4 *Instance Variables and Class Variables* if you missed either error.

Question 16

A. Named data like `speedLimit` and `lengthInMiles` is data abstraction: the program works with what the data means. Calling `fastestRoute` knowing only what it does is procedural abstraction.

- **B** is wrong: keeping every detail is the opposite of abstraction, and storing photos uses neither kind of abstraction for route-finding.
- **C** is wrong: copying steps everywhere is duplicated code. Procedural abstraction would put those steps in one named method and call it.
- **D** is wrong: unnamed numbers hide what the data means, which is the opposite of data abstraction.

Reread 3.1.2 *Data Abstraction: Naming Data* and 3.1.5 *Procedural Abstraction and Method Decomposition* if you chose differently.

Question 17

The program prints:

```
Hatchet: 9 days
Wonder: 10 days
```

Error: line 6 recomputes the days with its own copy of the calculation, `310 / 30`, instead of calling the method. `int` division gives 10 and drops the leftover 10 pages, so the round-up step that lives inside `daysToFinish` is missing. This is the duplicated-code problem: the rule was written in two places, and one copy is incomplete.

Corrected line 6:

```
1 int wonderDays = daysToFinish(310, 30);
```

Now `daysToFinish` computes $310 / 30 = 10$ with a remainder of 10, adds 1, and returns 11, so the program prints `Wonder: 11 days`. Reusing the method keeps the rule in one place.

Reread 3.1.6 *Generalizing with Parameters* if you did not spot the copied calculation.

Question 18

B. The signature is unchanged, and the new body returns the same rounded-up number of days as the old one (for example, 9 for 250 and 30, and 3 for 90 and 30). Callers depend only on the signature and what the method does, and both are preserved.

- **A** is wrong, and sneakily so: both parameters are `int`, so the signature's types (`int, int`) are unchanged and calls still compile. But existing calls such as `daysToFinish(250, 30)` would now put 250 into `pagesPerDay` and 30 into `pages`, so the method would no longer do what callers expect. What it does changed.
- **C** is wrong: the signature is the same, but what the method does is different, so every caller would get wrong answers with no error.
- **D** is wrong: a third parameter changes the signature, so every existing two-argument call would fail to compile.

Reread 3.1.7 *Changing How a Method Works* if you chose differently.

Question 19

Model answer:

(a) The locker number, the combination, and whether it is assigned are **instance variables**, because each locker has its own value for them: locker 12 and locker 13 have different numbers and combinations, and one can be assigned while the other is not. The height is a **class variable**, because every locker has the same height, so one shared value is enough, and changing it once updates every locker.

(b) The assign behavior needs the student's name, a `String` parameter, because the description says lockers are assigned "by name." The report behavior answers a yes-or-no question, so it returns a `boolean`.

A complete answer must:

- mark only the height as a class variable, using the "same value for every object?" test, and
- name a `String` parameter for the student's name and a `boolean` return value.

(Remembering the student's name as a fifth attribute is also a good decision, since the locker may need to know who it belongs to.)

Reread 3.1.8 *Designing a Class Before Coding* and 3.1.4 *Instance Variables and Class Variables* if your answer differed.

Question 20

(a) Both print:

```
10
55
```

For the loop: `sumTo(4)` adds $1 + 2 + 3 + 4 = 10$. For the formula: $4 \times 5 / 2 = 20 / 2 = 10$, and $10 \times 11 / 2 = 110 / 2 = 55$.

(b) The loop runs for $k = 1, 2, \dots, 10$, so `total += k`; runs **10 times**. Using the statement execution counts from Topic 2.12, the loop version does n additions for `sumTo(n)`, while the formula version does the same small amount of work for any n . For large n , the formula is much faster.

(c) Yes. The signature is still `sumTo(int)`, and both versions return the same result for these inputs and for any positive `n` small enough that the values stay inside the `int` range from Topic 1.5. The change only affects how the method works inside (it does far less work), which is exactly the kind of change callers do not need to know about.

Reread 3.1.7 *Changing How a Method Works* if your answer differed.

Question 21

C. As a class variable, `playsToday` has one value shared by every song. Each of the three plays adds 1 to that same value: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$. Song Y reports the shared value, 3, which is the total plays of all songs, not its own.

- **A** is wrong: 1 would be correct if `playsToday` were an instance variable, which is what the designer intended. The mistake changes the answer.
- **B** is wrong: 2 is Song X's own play count, but neither song keeps its own count when the variable is shared.
- **D** is wrong: Song Y was played, and every play changes the one shared value.

The fix is to make `playsToday` an instance variable, since each song needs its own count.

Reread 3.1.4 *Instance Variables and Class Variables* if you chose differently.