

1.12 Objects: Instances of Classes

Self-Study Guide · About 45 minutes · With answer key

By the end you will be able to

- Explain how a class and an object are related, using the `String` class as an example.
- Tell the attributes of an object (what it knows) apart from its behaviors (what it can do).
- Read a class hierarchy and say which attributes and behaviors a subclass gets from its superclass.
- Explain why every class in Java is a subclass of the `Object` class.
- Declare variables of a reference type and explain that they hold a reference to an object, not the object itself.
- Predict what happens when a reference variable is reassigned or copied.

What to remember from earlier

- Every variable has a name and a data type, either primitive or reference. ([Topic 1.2: Variables and Data Types, 1.2.2 Primitive Types and Reference Types](#))
- A string literal is text in double quotes, such as `"Ada"`. ([Topic 1.3: Expressions and Output, 1.3.3 Literals and String Literals](#))
- The assignment operator `=` stores the value on its right in the variable on its left. ([Topic 1.4: Assignment Statements and Input, 1.4.1 The Assignment Operator](#))
- The literal `null` means no object. ([Topic 1.4: Assignment Statements and Input, 1.4.5 Reference Variables and null](#))
- You call `Math` methods through the class name, as in `Math.sqrt(25.0)`. ([Topic 1.10: Calling Class Methods, 1.10.2 Calling with the Class Name and a Dot](#))

1.12.1 Classes and Objects

You have been using the word `String` since you first stored text in a variable, and you have read the `String` documentation. But what *is* `String`? It is not a piece of text itself: `"Ada"` is a piece of text, and so is `"Grace Hopper"`. Both are strings, yet they are different from each other. Java needs a way to describe "what every piece of text is like" separately from "this particular piece of text." That is the difference between a class and an object.

Start with an everyday picture. A cookie cutter is not a cookie. It describes a shape, and you can press it into dough again and again to make as many cookies as you like. Every cookie has the same shape, because it came from the same cutter, but each cookie is its own separate thing: one can have sprinkles, another chocolate chips. The cutter is the plan; the cookies are the things made from the plan.

In Java, the plan is a **class**: the written-out definition, or **blueprint**, of what a certain kind of thing knows and what it can do. A thing made from that plan is an **object**. Each object is one specific **instance** of its class, so "an instance of `String`" just means "one particular `String` object."

A class describes two kinds of things about its objects:

- **Attributes** are what each object knows about itself: its data. A `String` object knows the characters it contains, and therefore how many characters it has.
- **Behaviors** are what each object can do: its actions, written as methods. A `String` object can report its length, among other things you will meet in later topics.

Every object of a class has the **same kinds** of attributes, but each object has **its own values** for them. Every `String` object has characters; `"Ada"` has three of them and `"Grace Hopper"` has twelve.

KEY IDEA

A class is the blueprint that defines the attributes and behaviors of a kind of object. An object is one specific instance of a class, with its own values for those attributes.

The cookie-cutter picture stops working in one place. A real cutter only controls shape, so cookies can differ in ways the cutter never mentioned. A Java object can only have the attributes and behaviors its class defines. If the class does not describe it, the object does not have it.

WORKED EXAMPLE Three String objects, one class

This code stores three different pieces of text and then asks each one for its length. The method `length()` is a behavior of every `String` object: it gives back the number of characters in that string. (How to call a method on an object is taught fully in Topic 1.14; here, read `first.length()` as "ask the object that `first` refers to for its length.")

```
1 String first = "Ada";
2 String second = "Grace Hopper";
3 String third = "Alan";
4 System.out.println(first.length());
5 System.out.println(second.length());
6 System.out.println(third.length());
```

Look at it through the class/object lens:

Text in the code	Class or object?	Its own attribute value (characters)	Result of <code>length()</code>
<code>String</code>	Class	(none: it is the plan, not a string)	(not asked)
<code>"Ada"</code>	Object	A, d, a	3
<code>"Grace Hopper"</code>	Object	G, r, a, c, e, space, H, o, p, p, e, r	12
<code>"Alan"</code>	Object	A, l, a, n	4

The program displays:

```
3
12
4
```

All three objects have the same behavior, `length()`, because all three come from the `String` class. The answers differ because each object has its own attribute values: the space in `"Grace Hopper"` counts as a character, which is why the answer is 12. One class, one behavior, three different results.

The same pattern shows up in classes you have already used. `Scanner` (Topic 1.4) is a class; the thing in your program that actually reads what the user types is a `Scanner` object. And it works for things outside Java, too. Imagine a music app that describes songs:

Part of the plan	Song class defines	One object: a song	Another object: a song
Attributes	a title, an artist, a length	"Morning Light", Ana Ruiz, 3:12	"City Rain", Leo Park, 4:05
Behaviors	play, pause, report its title	can play, pause, report	can play, pause, report

The `Song` class is invented for this example, but the reasoning is the same as for `String`: one class, many objects, same kinds of attributes, different values, same behaviors.

WATCH OUT

Calling the class "the object." Students often say "`String` is an object." It is not; `String` is the class. "`Ada`" is an object, an instance of `String`. A handy check: if you can ask "which one?", you are talking about an object (which string? the one that says "`Ada`"). The class has only one definition, so "which `String` class?" makes no sense.

A related mistake is thinking objects of the same class must be identical. They share the same *kinds* of attributes and the same behaviors, but each object holds its own values.

PAUSE AND TRY

Question 1 Multiple choice

A program works with the text "`Biscuit`" and the text "`Pepper`", both stored as `Strings`. Which statement is correct?

- A. "`Biscuit`" is a class, and `String` is one of its objects.
- B. `String` is a class, and "`Biscuit`" and "`Pepper`" are two instances of it.
- C. "`Biscuit`" and "`Pepper`" are the same object, because they have the same class.
- D. "`Biscuit`" and "`Pepper`" are two different classes, because they hold different text.

Question 2 Short answer

A school app has a `Student` class. Sort these into **attributes** and **behaviors**: *grade level, sign in to class, student ID number, submit homework, name*. Then explain how two `Student` objects can both have a name attribute, yet one holds "`Maya`" and the other holds "`Jordan`."

1.12.2 Class Hierarchies and Inheritance

Suppose a team is writing classes for a car-rental company. They need a `Car` class and a `Truck` class. Both need a license plate, a number of miles driven, and a behavior to record a trip. If they write those three things in `Car` and then write them all again in `Truck`, they have two copies of the same code. When the company decides to track fuel level too, someone has to add it in two places, and sooner or later one copy gets forgotten.

The fix is to write the shared parts once, in one class, and let the other classes use them. In everyday words: describe what all vehicles have in common in one place, then describe only what is *different* about cars and trucks.

That shared class is called a **superclass**. A class that builds on a superclass is a **subclass**; programmers say the subclass **extends** the superclass. A subclass can use all of the superclass's attributes and behaviors **without writing them again**, and it can add its own. This connection, where a subclass draws on what its superclass already defines, is called **inheritance**. A set of classes linked this way is a **class hierarchy**.

KEY IDEA

A superclass holds the attributes and behaviors that related classes share. Each subclass extends the superclass, gets all of those without repeating them in its code, and can add its own.

WORKED EXAMPLE A vehicle hierarchy

The rental team puts the shared parts in a superclass called `Vehicle`. `Car` and `Truck` are subclasses of `Vehicle`.

Class	Relationship	Written in this class
<code>Vehicle</code>	Superclass	license plate, miles driven; behavior: record a trip
<code>Car</code>	Subclass of <code>Vehicle</code>	number of seats
<code>Truck</code>	Subclass of <code>Vehicle</code>	cargo capacity; behavior: load cargo

Now work out everything a `Truck` object has. Step 1: start with what `Truck` itself defines: cargo capacity and the load-cargo behavior. Step 2: follow the relationship up to `Vehicle` and add what it defines: license plate, miles driven, and record a trip. So a `Truck` object has **three attributes** (license plate, miles driven, cargo capacity) and **two behaviors** (record a trip, load cargo), even though the `Truck` class only wrote one attribute and one behavior.

Do the same for a `Car` object: number of seats from `Car`, plus license plate, miles driven, and record a trip from `Vehicle`.

What does a `Car` object **not** have? Cargo capacity. That belongs to `Truck`, and `Car` and `Truck` are not each other's superclass; they only share `Vehicle`.

Now change one thing. The company decides every vehicle must track its fuel level. The team adds a fuel-level attribute to `Vehicle` only. Nothing in `Car` or `Truck` is edited, yet every `Car` object and every `Truck` object now has a fuel level, because both draw on `Vehicle`. That is the payoff of inheritance: a change written once in the superclass reaches every subclass.

The same shape appears whenever related things share a core. A school might have a `Person` superclass with a name and an email address, and subclasses `Student` (adds grade level) and `Teacher` (adds room number). A student and a teacher both have a name and an email, because both are people.

A good way to check a hierarchy is the phrase "is a": a truck **is a** vehicle, and a student **is a** person. If the sentence sounds right, the subclass/superclass pair makes sense. "A vehicle is a

truck" sounds wrong, because not every vehicle is a truck, which is exactly why `Vehicle` must not hold truck-only features.

In this course you need to read hierarchies like these and explain what each class gets. Designing and writing your own superclasses and subclasses is not part of AP Computer Science A.

WATCH OUT

Reading inheritance backward. Inheritance goes one way: from the subclass up to the superclass. A subclass gets everything in its superclass, but a superclass gets nothing from its subclasses. Students sometimes reason "Truck has cargo capacity and Truck is related to Vehicle, so a Vehicle has cargo capacity." It does not. Always start at the class you are asked about and move **up** the hierarchy, never down or sideways.

PAUSE AND TRY

Question 3 Multiple choice

A game has three classes: `Dog`, `Cat`, and `Animal`. Every dog and cat has a name and an age and can make a sound. Only dogs can fetch, and only cats can climb. Which design uses a class hierarchy well?

- A. `Dog` is the superclass; `Cat` and `Animal` are subclasses. `Dog` holds name, age, make a sound, and fetch.
- B. `Animal` is the superclass; `Dog` and `Cat` are subclasses. `Animal` holds name, age, and make a sound; `Dog` adds fetch and `Cat` adds climb.
- C. `Animal` is the superclass and holds name, age, make a sound, fetch, and climb.
- D. No superclass; `Dog` and `Cat` each write name, age, and make a sound themselves.

Question 4 Short answer

Using the school hierarchy (`Person` holds name and email; `Student` adds grade level; `Teacher` adds room number), list every attribute a `Teacher` object has. Then explain why a `Student` object has no room number.

1.12.3 Every Class Is a Subclass of Object

The `Vehicle` hierarchy had a class at the top. Does `Vehicle` itself have a superclass? Does `String`? Java answers this the same way for every class, and the answer explains a line you may have skimmed past in the `String` documentation in Topic 1.7.

Java comes with a class named `Object`, with a capital O. It sits at the very top of every class hierarchy: **every class in Java is a subclass of `Object`**. When a class names no other superclass, Java makes `Object` its superclass automatically. When it does name one, like `Truck` naming `Vehicle`, then `Object` is still above it, one or more levels up. Either way, following the chain upward from any class always ends at `Object`.

Why would Java do this? Because there are a few behaviors every object in a program should have, no matter what kind it is. Java writes them once, in `Object`, and every class draws on them through inheritance, exactly as `Car` draws on `Vehicle`.

KEY IDEA

Every class in Java is a subclass of the `Object` class, directly or through other superclasses. So every object can use the behaviors that `Object` defines.

WORKED EXAMPLE Finding `Object` in the `String` documentation

Near the top of the `String` documentation page (Topic 1.7), above the description of the class, is this short chain:

```
java.lang.Object
  java.lang.String
```

Read it top to bottom as a hierarchy. `java.lang.Object` is the superclass; the indented `java.lang.String` below it is the subclass. (The `java.lang.` part is the name of the library both classes belong to.) So `String` is a subclass of `Object`. The `Scanner` documentation shows the same pattern, with `java.util.Scanner` indented under `java.lang.Object`.

Further down the `String` page is a section titled "Methods declared in class `java.lang.Object`." It lists behaviors such as `getClass`, `notify`, and `wait`. Nobody wrote those in the `String` class; `String` objects have them because `String` draws on its superclass.

Here is one of them in action. `getClass()` is a behavior from `Object` that reports which class an object belongs to. (It is not on the AP exam; it is used here only because it lets you see an inherited behavior work.)

```
1 String firstWord = "Ada";
2 String secondWord = "Grace";
3 System.out.println(firstWord.getClass());
4 System.out.println(secondWord.getClass());
```

```
class java.lang.String
class java.lang.String
```

Both objects can use `getClass()`, even though the `String` class never defines it, because every `String` is also an `Object`. Both report the same class, because both are instances of `String`: different values, same blueprint.

Now vary the hierarchy. In the school example, `Person` names no superclass, so its superclass is `Object`. `Student` extends `Person`. Follow the chain up from `Student`:

Class	Its superclass	How <code>Object</code> is reached
<code>Object</code>	(none: it is the top)	It is <code>Object</code>
<code>Person</code>	<code>Object</code>	Directly, one level up
<code>Student</code>	<code>Person</code>	Through <code>Person</code> , two levels up

So `Student` is a subclass of `Person` **and**, one more level up, of `Object`. A `Student` object has `Student`'s attributes, everything from `Person`, and the behaviors from `Object`.

WATCH OUT

Mixing up `Object` and `object`. Lowercase **object** is the general word for any instance of any class: `"Ada"` is an object. Capitalized `Object` is the name of one specific class, the one at the top of every hierarchy. Java is case-sensitive (Topic 1.1), so these are different words. When you read `"String is a subclass of Object,"` it names a class; when you read `"Ada is an object,"` it describes an instance.

A second mistake: thinking only classes that say so have a superclass. Every class has one. If none is named, it is `Object`.

PAUSE AND TRY**Question 5** Multiple choice

In a library app, `Item` names no superclass, and `Book` extends `Item`. Which statement is true?

- A. `Book` is a subclass of `Item` but not of `Object`, because it already has a superclass.
- B. `Item` has no superclass, because none was named.
- C. `Book` is a subclass of `Item`, and both `Item` and `Book` are subclasses of `Object`.
- D. `Object` is a subclass of `Book`, because `Book` is at the bottom.

Question 6 Short answer

A classmate reads the sentence "Every `String` is an object, and `String` is a subclass of `Object`" and says it repeats itself. Explain what each of the two words, *object* and `Object`, means in that sentence.

1.12.4 Reference Variables

In Topic 1.2 you learned that a variable is a storage location with a name and a type, and that types are either primitive or reference. A primitive variable is simple: `int score = 95;` puts the number 95 in the box named `score`. But an object can be large. A `String` might hold a three-letter name or every word of a novel. Java cannot make every variable box big enough for any object. So how does a variable "hold" an object?

It does not hold the object at all. Think of a phone contact. Your contacts list does not contain your friend; it contains your friend's number, which lets you reach them. The object lives somewhere in the computer's memory, and the variable holds a way to find it.

A type that names a class, such as `String` or `Scanner`, is a **reference type**. A variable of a reference type holds an **object reference**: information that tells Java where the object is. You can picture a reference as the **memory address** of the object, like a street address that leads to a house. The object itself is stored at that address, outside the variable.

KEY IDEA

A primitive variable holds its value directly. A variable of a reference type holds a reference to an object, which you can think of as the object's memory address, not the object itself.

To declare a variable that can refer to an object, use the class name as the type:

```
1 String name = "Ada";
```

Piece	What it is
String	The type: a reference type, named by the class. Class names start with a capital letter.
name	The variable's name, as in Topic 1.2.
=	Assignment, as in Topic 1.4: store the value on the right in the variable on the left.
"Ada"	A string literal (Topic 1.3). It gives Java a String object; name gets a reference to it.

Declaring `String name;` with no assignment is also allowed, as long as the variable is assigned before it is used (Topic 1.4). And as you saw in Topic 1.4, a reference variable can hold `null`, meaning it refers to no object at all. Creating objects in other ways is the subject of Topic 1.13.

WORKED EXAMPLE A primitive and a reference side by side

```
1 int score = 95;
2 String name = "Ada";
3 System.out.println(score);
4 System.out.println(name);
```

Here is what memory looks like after the first two lines. The address #5021 is made up for this picture; real addresses are chosen by Java and you never write them yourself.

Variable	Type	What its box holds	Where that leads
score	int	95	(nowhere: the value is here)
name	String	a reference, like #5021	the String object "Ada"

The program displays:

```
95
Ada
```

Why does the second line show `Ada` and not `#5021`? Because `System.out.println` does not display the reference; it follows the reference to the object and displays the object's text. The address stays hidden. That is convenient, but it can fool you into thinking `name` contains the letters A, d, a. It contains only the way to find them.

Now change one thing: replace `"Ada"` with the full text of a 300-page novel. The `"Ada"` object was tiny and the novel is huge, but `name`'s box does not grow. It still holds just a reference; only the object at the other end of it is bigger. This is why every variable can have a fixed-size box: a primitive's value fits in it, and for an object only the reference has to.

The type in a declaration must match the kind of value assigned. A reference variable of type `String` can refer only to a `String` object, and an `int` variable can hold only an `int`. Java checks this at compile time.

WATCH OUT

Two type mistakes in declarations. This program has two errors:

WrongType.java

```
1 public class WrongType
2 {
3     public static void main(String[] args)
4     {
5         string city = "Boston";
6         int name = "Ada";
7         System.out.println(city);
8         System.out.println(name);
9     }
10 }
```

WrongType.java:5: error: cannot find symbol

WrongType.java:6: error: incompatible types: String cannot be converted to int

- Line 5 writes `string` with a lowercase `s`. Java is case-sensitive, and there is no class named `string`, so Java cannot find it. Students make this mistake because `int` and `double` are lowercase; class names like `String` are capitalized. Fix:
`String city = "Boston";`
- Line 6 tries to put a reference to a `String` object into an `int` variable. An `int` box holds a whole number, not a reference to text. Fix: declare it as `String name = "Ada";`.

Both are compiler errors, so nothing runs until both are fixed.

PAUSE AND TRY**Question 7** Debugging

This program should display a building name and a room number, but it does not compile. Find the error, explain what the compiler reports and why, and write a corrected line.

RoomNumber.java

```
1 public class RoomNumber
2 {
3     public static void main(String[] args)
4     {
5         String building = "North Hall";
6         String roomNumber = 214;
7         System.out.println(building);
8         System.out.println(roomNumber);
9     }
10 }
```

Question 8 Multiple choice

After `String city = "Boston";` runs, what does the variable `city` hold?

- A. The six characters B, o, s, t, o, n
- B. A reference that tells Java where the `String` object `"Boston"` is stored

- C. The `String` class
- D. The number 6, the length of the text

1.12.5 Assigning and Copying References

In Topic 1.4 you learned that assigning a new value to a variable replaces the old one, and that `a = b`; copies the value of `b` into `a`. Those rules do not change for reference variables. What changes is *what* gets replaced or copied: the reference, not the object.

In plain words: if you write a new address on a sticky note, the note now leads to a different house, but the old house is still standing. If you copy an address onto a second note, you have two notes and still only one house.

KEY IDEA

Assigning to a reference variable replaces the reference it holds, so it now refers to a different object. Copying a reference variable copies the reference, so both variables refer to the same object; no new object is made.

WORKED EXAMPLE Copying a reference, then reassigning

Predict the two lines of output before reading on.

```
1 String name = "Ada";
2 String backup = name;
3 name = "Grace";
4 System.out.println(name);
5 System.out.println(backup);
```

Trace it one statement at a time. As before, the addresses are made up to make the references visible.

After line	name holds	backup holds	Objects in memory
1	#5021 (→ Ada)	(not declared)	"Ada" at #5021
2	#5021 (→ Ada)	#5021 (→ Ada)	"Ada" at #5021
3	#7340 (→ Grace)	#5021 (→ Ada)	"Ada" at #5021, "Grace" at #7340

- **Line 1:** `name` gets a reference to the "Ada" object.
- **Line 2:** `backup = name` copies what `name` holds, the reference #5021. Now two variables refer to the **same** object. There is still only one "Ada".
- **Line 3:** `name` gets a reference to a different object, "Grace". Only `name`'s box changes. `backup` still holds #5021, and the "Ada" object is untouched.

The program displays:

```
Grace
Ada
```

Now change one thing: swap lines 2 and 3, so `name` is reassigned **before** the copy. Then `backup = name` copies #7340, and both variables refer to "Grace"; the output is Grace twice. The statements are identical, only the order differs (sequencing, Topic 1.1), and the result changes because the copy takes whatever reference `name` holds at that moment.

WATCH OUT

"Changing `name` changes `backup` too." Students who learn that two variables can refer to the same object sometimes conclude that the two variables are now linked forever. They are not. Each variable has its own box. `backup = name` copied a reference once; reassigning `name` afterward replaces only `name`'s reference. To predict output, track what each box holds after each line, as in the trace table above.

PAUSE AND TRY

Question 9 Tracing

Write the output of this code. Then answer: when the last line runs, does `day` still refer to the "Monday" object, and did the reassignments change that object?

```
1 String day = "Monday";
2 System.out.println(day);
3 day = "Tuesday";
4 day = "Friday";
5 System.out.println(day);
```

Putting it together

This program uses every idea in the chapter. Read it, predict the four lines of output, then check the trace.

HopperFacts.java

```
1 public class HopperFacts
2 {
3     public static void main(String[] args)
4     {
5         String fullName = "Grace Hopper";
6         String job = "computer scientist";
7         String hero = fullName;
8         fullName = "Admiral Hopper";
9         System.out.println(hero);
10        System.out.println(fullName);
11        System.out.println(job.length());
12        System.out.println(hero.getClass());
13    }
14 }
```

Classes and objects (1.12.1). `String` is the class. The program makes three `String` objects from literals: `"Grace Hopper"`, `"computer scientist"`, and `"Admiral Hopper"`. Each has its own characters.

Reference variables (1.12.4, 1.12.5). Track the boxes, using made-up addresses:

After line	<code>fullName</code>	<code>job</code>	<code>hero</code>
5	#100 (→ Grace Hopper)	(not declared)	(not declared)
6	#100 (→ Grace Hopper)	#200 (→ computer scientist)	(not declared)
7	#100 (→ Grace Hopper)	#200	#100 (→ Grace Hopper)
8	#300 (→ Admiral Hopper)	#200	#100 (→ Grace Hopper)

Line 7 copies the reference, so `hero` and `fullName` briefly refer to the same object. Line 8 reassigns only `fullName`; `hero` keeps its reference to `"Grace Hopper"`.

Behaviors (1.12.1). Line 11 asks the `"computer scientist"` object for its length: 8 letters in `"computer"`, 1 space, 9 letters in `"scientist"`, for 18.

Hierarchy (1.12.2, 1.12.3). Line 12 uses `getClass()`, which `String` objects have because `String` is a subclass of `Object`.

```
Grace Hopper
Admiral Hopper
18
class java.lang.String
```

Vocabulary

Term	Definition	Example
class	The blueprint that defines the attributes and behaviors of a kind of object	<code>String</code>
object / instance	One specific thing made from a class, with its own attribute values	<code>"Ada"</code>
attribute	Data each object of a class has	The characters of a string
behavior	An action each object of a class can do, written as a method	<code>length()</code>
superclass	A class that holds attributes and behaviors shared by related classes	<code>Vehicle</code>
subclass	A class that extends a superclass and gets its attributes and behaviors	<code>Truck</code> extends <code>Vehicle</code>
inheritance	The relationship in which a subclass draws on what its superclass defines	A <code>Truck</code> has a license plate from <code>Vehicle</code>
class hierarchy	A set of classes connected by superclass/subclass relationships	<code>Object</code> → <code>Person</code> → <code>Student</code>
Object	The class at the top of every hierarchy; every class is a subclass of it	<code>String</code> is a subclass of <code>Object</code>

reference type	A data type named by a class; its variables hold references to objects	String, Scanner
object reference	What a reference variable holds; think of it as the object's memory address	name holds a reference to "Ada"

Chapter summary

- A **class** is a blueprint that defines **attributes** (what objects know) and **behaviors** (what they can do). An **object** is one **instance** of a class with its own attribute values.
- A **superclass** holds what related classes share. A **subclass** extends it and, through **inheritance**, gets those attributes and behaviors without rewriting them, and can add its own. Inheritance goes from subclass up to superclass only.
- Every class in Java is a subclass of the `Object` class, directly or through other superclasses, so every object has the behaviors `Object` defines.
- A variable of a **reference type**, such as `String`, holds an **object reference** (picture a memory address), not the object. Its type must match the object's class.
- Assigning to a reference variable replaces its reference; copying one copies the reference, so both variables refer to the same object until one is reassigned.

Review questions

Question 10 Multiple choice

A bank app has a superclass `Account` (attributes: owner name, balance). `SavingsAccount` and `CheckingAccount` are subclasses of `Account`. `SavingsAccount` adds an interest rate; `CheckingAccount` adds a monthly fee. Which statement is true?

- A `CheckingAccount` object has an interest rate, because it is related to `SavingsAccount`.
- An `Account` object has a monthly fee, because its subclass `CheckingAccount` defines one.
- A `SavingsAccount` object has an owner name, a balance, an interest rate, and the behaviors defined in `Object`.
- `CheckingAccount` is not a subclass of `Object`, because its superclass is `Account`.

Question 11 Short answer

A classmate says: "'Hello' and 'Hi' are both `String` objects, so they have the same attributes and `length()` must give the same answer for both." Explain what is right and what is wrong in this statement.

Question 12 Short answer

In the vehicle hierarchy (`Vehicle` superclass; `Car` and `Truck` subclasses), the company now wants every vehicle to be able to report its next service date. A programmer plans to add that behavior to `Car` and to `Truck`. Suggest a better plan and give two reasons it is better.

Question 13 Multiple choice

A program needs a variable named `keyboard` that can refer to a `Scanner` object. Which declaration is correct?

- A. `scanner keyboard;`
- B. `String keyboard;`
- C. `Scanner keyboard;`
- D. `int keyboard;`

Question 14 Debugging

A music app should display the song that was playing before the change, then the song playing now:

```
Morning Light
City Rain
```

This code compiles and runs, but displays `City Rain` twice. Explain the logic error using references, and give the fix.

```
1 String nowPlaying = "Morning Light";
2 nowPlaying = "City Rain";
3 String previous = nowPlaying;
4 System.out.println(previous);
5 System.out.println(nowPlaying);
```

Reflection

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

- ☐ I can explain the difference between a class and an object, using `String` as an example. Reread: 1.12.1 Classes and Objects
- ☐ I can sort features of a class into attributes and behaviors. Reread: 1.12.1 Classes and Objects
- ☐ I can list everything a subclass gets from its superclass, and what it does not get. Reread: 1.12.2 Class Hierarchies and Inheritance
- ☐ I can explain why every class, including `String`, is a subclass of `Object`. Reread: 1.12.3 Every Class Is a Subclass of `Object`
- ☐ I can declare a reference variable and explain that it holds a reference, not the object. Reread: 1.12.4 Reference Variables
- ☐ I can trace what each reference variable refers to after assignments and copies. Reread: 1.12.5 Assigning and Copying References

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. A class is the blueprint; each object is one specific instance of it with its own attribute values. `String` is the blueprint, and each piece of text is a separate `String` object.

- **A** is wrong: it reverses the relationship. The class is the general plan (`String`); the specific text is the object.
- **C** is wrong: sharing a class means sharing the same kinds of attributes and behaviors, not being the same object. Each holds its own characters.
- **D** is wrong: different values do not make different classes. Both are made from the same blueprint, `String`; they just have different attribute values.

Reread 1.12.1 *Classes and Objects* if you chose differently.

Question 2

Model answer: Attributes (what a student object knows): grade level, student ID number, name. Behaviors (what a student object can do): sign in to class, submit homework. The `Student` class says every student object *has* a name, so both objects have that attribute. But each object is a separate instance with its own values, so the class decides that a name exists while each object decides what its name is.

A complete answer must:

- put the three pieces of data under attributes and the two actions under behaviors, and
- explain that the class defines which attributes exist, while each object has its own values for them.

Reread 1.12.1 *Classes and Objects* if your sorting or explanation differed.

Question 3

B. A superclass should hold exactly what its subclasses share (name, age, make a sound). Each subclass adds only what is special about it. A dog *is an* animal and a cat *is an* animal, so `Animal` belongs on top.

- **A** is wrong: a cat is not a dog, and every cat would get fetch from `Dog`. The "is a" check fails.
- **C** is wrong: putting fetch and climb in `Animal` gives every subclass both, so cats could fetch and dogs could climb. A superclass holds only shared features.
- **D** is wrong: it works, but it writes the shared parts twice, which is the repetition a hierarchy is meant to remove.

Reread 1.12.2 *Class Hierarchies and Inheritance* if you chose differently.

Question 4

Model answer: A `Teacher` object has a room number (from `Teacher`) plus a name and an email (from its superclass `Person`): three attributes. A `Student` object has no room number because room number is written in `Teacher`, and `Teacher` is not a superclass of `Student`. A subclass only gets what is in its own class and its superclass.

A complete answer must:

- list room number, name, and email, saying where each comes from, and
- explain that inheritance goes from a subclass up to its superclass only, and `Student` and `Teacher` only share `Person`.

Reread 1.12.2 *Class Hierarchies and Inheritance* if your list or reasoning differed.

Question 5

C. Every class is a subclass of `Object`. `Item` names no superclass, so its superclass is `Object`. `Book` extends `Item`, so `Object` is two levels above `Book`.

- **A** is wrong: having a named superclass does not cut a class off from `Object`. The chain continues upward past `Item` to `Object`.
- **B** is wrong: when no superclass is named, Java makes `Object` the superclass.
- **D** is wrong: it reverses the direction. `Object` is at the top of every hierarchy, so it is a superclass of everything, never a subclass.

Reread 1.12.3 *Every Class Is a Subclass of Object* if you chose differently.

Question 6

Model answer: Lowercase "object" means an instance of a class: each piece of text such as "Ada" is an object made from the `String` blueprint. Capitalized `Object` is the name of a particular class, the one at the top of every Java class hierarchy. The sentence says two different things: `String` values are instances of a class, and the `String` class sits below the `Object` class in the hierarchy.

A complete answer must:

- define lowercase object as an instance of a class, and
- identify `Object` as the specific class every class is a subclass of.

Reread 1.12.3 *Every Class Is a Subclass of Object* if your explanation differed.

Question 7

Error: line 6 declares `roomNumber` as a `String` but assigns the `int` value 214. A `String` variable holds a reference to a `String` object, and 214 (no quotes) is a primitive `int`, not a `String` object. The compiler reports:

```
RoomNumber.java:6: error: incompatible types: int cannot be converted to String
```

Corrected line (make the value a string literal):

```
1 String roomNumber = "214";
```

The program then displays North Hall and 214:

```
1 String building = "North Hall";
2 String roomNumber = "214";
3 System.out.println(building);
4 System.out.println(roomNumber);
```

Declaring it as `int roomNumber = 214;` would also compile and display the same output. Either way, the type and the value must match.

Reread 1.12.4 *Reference Variables* if you missed the error.

Question 8

B. `city` has a reference type (`String`), so it holds a reference to the object, which you can picture as the object's memory address. The object, with its characters, is stored elsewhere.

- **A** is wrong: this describes the object, not the variable. Printing `city` shows the characters only because `println` follows the reference.
- **C** is wrong: the class is the blueprint. `city` refers to one object made from it.
- **D** is wrong: the length is something the object can report, not what the variable stores.

Reread 1.12.4 *Reference Variables* if you chose differently.

Question 9

After line	day refers to
1	"Monday"
3	"Tuesday"
4	"Friday"

Line 2 displays `Monday`. Lines 3 and 4 each give `day` a new reference, so by line 5 it refers to `"Friday"`:

```
Monday
Friday
```

No, `day` no longer refers to `"Monday"`; and no, the `"Monday"` object itself was never changed. Each assignment only replaced the reference in `day`'s box, so `day` was pointed at a different object each time. `"Tuesday"` was never displayed, because `day` was reassigned again before any `println` ran.

Reread 1.12.5 *Assigning and Copying References* if your output differed.

Question 10

C. Start at `SavingsAccount` and move up: it defines interest rate, gets owner name and balance from `Account`, and, because every class is a subclass of `Object`, gets `Object`'s behaviors too.

- **A** is wrong: `SavingsAccount` and `CheckingAccount` only share a superclass. Neither inherits from the other, so a checking account has no interest rate.
- **B** is wrong: this reads inheritance backward. A superclass gets nothing from its subclasses.
- **D** is wrong: `Account` names no superclass, so its superclass is `Object`, which puts `Object` above `CheckingAccount` too.

Reread 1.12.2 *Class Hierarchies and Inheritance* and 1.12.3 *Every Class Is a Subclass of Object* if you chose differently.

Question 11

Model answer: It is right that both are instances of the `String` class, so both have the same *kinds* of attributes (their characters) and the same behaviors, including `length()`. It is wrong that they have the same attribute *values*: each object holds its own characters. `length()` reports on the object it is asked about, so `"Hello"` gives 5 and `"Hi"` gives 2.

A complete answer must:

- agree that the same class means the same kinds of attributes and the same behaviors, and
- explain that each object has its own values, so the same behavior can give different results.

Reread 1.12.1 *Classes and Objects* if your explanation differed.

Question 12

Model answer: Add the behavior once, to `Vehicle`. Because `Car` and `Truck` are subclasses of `Vehicle`, both automatically get it without any change to their own code. This is better because (1) the code is written once instead of twice, so it cannot get out of sync if someone later changes one copy and forgets the other, and (2) any new subclass added later, such as `Van`, gets the behavior automatically too.

A complete answer must:

- place the behavior in the superclass, `Vehicle`, and
- give reasons based on inheritance: subclasses get it without rewriting it, so there is one copy to maintain and future subclasses get it as well.

Reread 1.12.2 *Class Hierarchies and Inheritance* if your plan differed.

Question 13

C. To declare a variable that refers to an object, use the object's class name as the type. `Scanner` is a reference type, so `keyboard` will hold a reference to a `Scanner` object once one is assigned (creating objects is Topic 1.13).

- **A** is wrong: Java is case-sensitive and class names are capitalized. There is no class named `scanner`, so the compiler reports `cannot find symbol`.
- **B** is wrong: a `String` variable can refer only to a `String` object. Assigning a `Scanner` object to it would be an incompatible-types error, just like the `String-into-int` error you saw.
- **D** is wrong: `int` is a primitive type. Its box holds a whole number, not a reference to any object.

Reread 1.12.4 *Reference Variables* if you chose differently.

Question 14

What goes wrong: trace the references.

After line	nowPlaying refers to	previous refers to
1	"Morning Light"	(not declared)
2	"City Rain"	(not declared)
3	"City Rain"	"City Rain"

Line 3 copies the reference `nowPlaying` holds **at that moment**, which already leads to "City Rain". Nothing refers to "Morning Light" any more, so it can never be displayed. Copying a reference does not remember earlier values.

Fix: copy the reference before reassigning, by moving `String previous = nowPlaying;` above the reassignment:

```
1 String nowPlaying = "Morning Light";
2 String previous = nowPlaying;
3 nowPlaying = "City Rain";
4 System.out.println(previous);
5 System.out.println(nowPlaying);
```

Now `previous` copies the reference to `"Morning Light"`, then only `nowPlaying` changes, and the output is `Morning Light` then `City Rain`. It was a logic error (Topic 1.1): the code followed every rule, so only comparing the output with the expected outcome revealed it.

Reread *1.12.5 Assigning and Copying References* if your explanation differed.