

1.12 Objects: Instances of Classes

Lesson Plan

Pacing: 2 class periods of 45 minutes

Learning objectives: 1.12.A, 1.12.B

Essential knowledge: 1.12.A.1, 1.12.A.2, 1.12.A.3, 1.12.B.1

Materials

- Projector and the Blueprints and Instances slide deck
- Whiteboard and markers
- Sticky notes (two colors) for the Day 2 memory picture
- IDE on every computer
- Printed Blueprints and Instances puzzle worksheet (answer key for you) and exit ticket

Agenda

Minutes	Activity
5	Day 1 warm-up — three pieces of text
10	Day 1 direct instruction — classes, objects, attributes, behaviors (slides 2–7)
6	Day 1 guided practice — class or object? (slide 8), then sort attributes and behaviors
10	Day 1 direct instruction — superclasses, subclasses, inheritance (slides 9–12)
6	Day 1 direct instruction — every class is a subclass of Object (slides 13–14)
6	Day 1 independent practice — puzzle worksheet questions 4–6
2	Day 1 closure — class or object?
5	Day 2 warm-up — what is in your contacts list?
9	Day 2 direct instruction — reference types and object references (slides 15–16)
5	Day 2 direct instruction — declarations whose type must match
9	Day 2 direct instruction — reassigning and copying references (slides 17–18)
9	Day 2 guided practice — track every box (slide 19) and the playlist bug
3	Day 2 closure — one blueprint, many objects (slides 20–21)
5	Day 2 exit ticket

Overview

Two periods on two learning objectives. Day 1 explains the relationship between a class and an object (1.12.A): a class is the blueprint that defines attributes and behaviors, and an object is one specific instance of it (1.12.A.1); related classes can share a superclass, and subclasses draw on it through inheritance (1.12.A.2); and every class in Java is a subclass of `Object` (1.12.A.3). Day 2 covers declaring variables of reference types (1.12.B): a reference variable holds an object

reference, which can be thought of as the object's memory address (1.12.B.1), so assigning or copying one replaces or copies the reference, not the object.

The CED's suggested skill is 4.A (describe the behavior of a code segment or program). Both days ask students to say what a class gives its objects, and to trace exactly what each variable refers to.

Stay inside the topic boundary:

- `String` is the running example. Every object in code comes from a string literal; creating objects with `new` is Topic 1.13. The only `new` students see is `new Scanner(System.in)`, the line they already use from Topic 1.4, which slide 8 asks them to label as a class or an object.
- Calls on a `String` object are limited to `length()`, which students met in Topic 1.7, and `getClass()`, used once to show an inherited behavior working. Say once that the rules for calling a method on an object are taught fully in Topic 1.14, and that `getClass` is not on the AP exam.
- Hierarchies are read, not written. Designing and implementing inheritance is outside the AP course (the CED's exclusion statement), so no code in this lesson uses `extends`. `Vehicle`, `Car`, `Truck`, and the other hierarchy classes are described in tables only.
- A label and a value still take a `print` followed by a `println`; joining text with `+` is Topic 1.15.
- Method headers in documentation start with `public`. Students can read past it; it is explained in a later topic.

TEACHER NOTE

The slides are the Blueprints and Instances deck (21 slides, with speaker notes); slide numbers below refer to it. Day 1 uses slides 1–14 and Day 2 uses slides 15–21. The Blueprints and Instances puzzle worksheet (9 questions, about 30 minutes) covers both days: Part B questions 4–6 need only Day 1 material, and questions 1–3 and 7–9 need Day 2.

Every program below is in the topic's `examples/` folder. Files whose names end in `.snippet.java` hold only the statements that go inside `main`; paste them into the `main` of a scratch class to run them.

- **Day 1, classes and objects:** `NameLengths`.
- **Day 1, every class is a subclass of `Object`:** `InheritedBehavior`.
- **Day 2, reference variables:** `PrimitiveAndReference` and `WrongType`.
- **Day 2, reassigning and copying:** `CopyReference` and `CopyReferenceSwapped`.
- **Day 2 guided practice:** `PlaylistBug` and `PlaylistFixed`.
- **Printed only (exit ticket):** `ExitMeals`.

Compiler messages are quoted from a recent JDK; the wording can vary slightly between Java versions. Before Day 1, open the `String` documentation page in a browser tab so you can scroll to the class chain at the top and the "Methods declared in class `java.lang.Object`" section.

Day 1 warm-up: three pieces of text

Write "Ada", "Grace Hopper", and "Alan" on the board. Ask pairs two questions: "What do these three have in common?" and "How is each one different from the others?" Give two minutes, then collect answers in two columns.

Expected: in common, all three are text in double quotes and all three are `Strings` (Topic 1.2); each has characters and a number of characters. Different: the actual characters, and so the count (3, 12, and 4, because the space in "Grace Hopper" counts). Leave both columns on the board. The left column will become "the class" and the right column "each object."

TEACHER NOTE

If a student says "they are all the same type," write it down and underline it. That sentence is today's first idea: one type, `String`, and many separate pieces of text made from it.

Day 1 direct instruction: classes, objects, attributes, behaviors

1. **The cookie cutter** (slide 2). Give pairs a minute on the slide's four questions, then draw out the analogy. A cutter is not a cookie. It describes a shape, and every cookie pressed from it has that shape, yet each cookie is a separate thing. The cutter is the plan; the cookies are made from the plan.
2. **Class and object** (1.12.A.1, slides 3–5). Read the five goals on slide 3; the first four are today's and the last is Day 2's. A **class** is the formal definition, or **blueprint**, of the attributes and behaviors of a kind of object. An **object** is one specific **instance** of a class. `String` is a class; "Ada" is an object, an instance of `String`. Point at the warm-up columns: the left column describes the class, the right column describes each object.
3. **Attributes and behaviors**. Students met both words in Topic 1.7. **Attributes** are what each object knows (its data): a `String` object knows its characters. **Behaviors** are what each object can do (its methods): a `String` object can report its `length()`. Every object of a class has the **same kinds** of attributes and the same behaviors, but each object has **its own values**.
4. **Run `NameLengths`** (slide 6 afterward). Before running it, say that `first.length()` asks the object `first` refers to for its number of characters; the rules for these calls are taught fully in Topic 1.14. Students predict 3, 12, 4. One class, one behavior, three different answers, because each object has its own characters. Slide 6 shows the same pattern with "Kai", "Marisol", and "Jo Ann" (3, 7, and 6, because the space counts).
5. **Where the picture breaks**. A real cutter controls only shape, so cookies can differ in ways the cutter never mentions. A Java object has only the attributes and behaviors its class defines.

MISCONCEPTION String is an object

Students say "String is an object" because they use the word for anything in a program. `String` is the class; "Ada" is an object. Slide 7 gives them a test: if "which one?" makes sense, it is an object ("which string? the one that says Ada"). "Which `String` class?" makes no sense, because there is one definition. Use the test on every example today.

MISCONCEPTION Objects of the same class are identical

Students hear "same class" and expect the same values. Point at the `NameLengths` output: the same behavior, `length()`, gives 3, 12, and 4. The class decides which attributes exist; each object holds its own values for them.

Day 1 guided practice: sort attributes and behaviors

Start with slide 8 (Class or object?). Pairs label each item, then check with the which-one test: `Scanner` and `String` are classes; `"pizza"`, the object `new Scanner(System.in)` creates, and `""` are objects. The empty string trips students up: it has no characters, but it is still one specific instance of `String`.

Then write the sorting task on the board. A library app has a `LibraryBook` class. Pairs sort the six items (title, check out, author, renew, due date, report whether it is overdue) into two columns, then answer the follow-up.

Attributes (what each book object knows)	Behaviors (what each book object can do)
title	check out
author	renew
due date	report whether it is overdue

Follow-up: "Two `LibraryBook` objects are on the cart. How can both have a title, yet one is *Hatchet* and the other is *Holes*?" Expected: the class says every book object has a title; each object is a separate instance with its own value.

TEACHER NOTE

A quick sorting check for students who are stuck: an attribute is a noun the object could tell you ("my due date is ..."); a behavior is a verb the object can do ("renew me"). "Report whether it is overdue" is a behavior even though it mentions the due date: it is an action that uses an attribute.

Day 1 direct instruction: superclasses, subclasses, inheritance

1. **The repetition problem** (slide 9). A car-rental team needs a `Car` class and a `Truck` class. Both need a license plate, miles driven, and a behavior to record a trip. Ask: "What goes wrong if they write those three things twice?" Expected: twice the work, and when one copy changes, the other gets forgotten.
2. **Superclass and subclass** (1.12.A.2, slide 10). Put the shared parts in one class, `Vehicle`, the **superclass**. `Car` and `Truck` are **subclasses**; each **extends** `Vehicle`. A subclass draws on the superclass's attributes and behaviors **without writing them again**, and adds its own. This relationship is **inheritance**, and a set of classes linked this way is a **class hierarchy**.
3. **Predict: what does a `Truck` object have?** Use the diagram on slide 10 (`Vehicle`: license plate, miles driven, record a trip; `Car`: number of seats; `Truck`: cargo capacity, load cargo). Pairs list

everything a `Truck` object has. Expected: three attributes (license plate, miles driven, cargo capacity) and two behaviors (record a trip, load cargo). Method: start at the class you are asked about and move **up**. Slide 11 reveals the answer step by step.

4. **What a `Car` does not have.** Cargo capacity: it is written in `Truck`, and `Car` and `Truck` only share `Vehicle`.
5. **The payoff** (slide 11). The company now tracks fuel level. Add it to `Vehicle` only, and every `Car` and every `Truck` object has a fuel level with no edits to `Car` or `Truck`.
6. **The "is a" check** (slide 10). A truck **is a** vehicle: the pair makes sense. "A vehicle is a truck" sounds wrong, which is why `Vehicle` must not hold truck-only features.
7. **The boundary.** In this course students read hierarchies like this and explain what each class gets. Designing and writing superclasses and subclasses is not part of AP Computer Science A.

MISCONCEPTION Reading inheritance backward or sideways

Students reason "`Truck` has cargo capacity, and `Truck` is related to `Vehicle`, so a `Vehicle` has cargo capacity," or give `Car` something from `Truck`. Inheritance goes one way, from a subclass up to its superclass (slide 12). Have students put a finger on the class they are asked about and move it only upward through the table, collecting what each class lists.

Day 1 direct instruction: every class is a subclass of `Object`

1. **The question.** "`Vehicle` is at the top of our table. Does it have a superclass? Does `String`?"
2. **The rule** (1.12.A.3, slide 13). Java has a class named `Object`, with a capital O. Every class in Java is a subclass of `Object`. A class that names no superclass gets `Object` as its superclass automatically; a class that names one, like `Truck`, still reaches `Object` one or more levels up.
3. **Show it in the documentation.** Scroll to the top of the `String` page (Topic 1.7). The chain `java.lang.Object` with `java.lang.String` indented below it is a hierarchy read top to bottom: `String` is a subclass of `Object`. Scroll down to "Methods declared in class `java.lang.Object`": `getClass`, `notify`, `wait`, and others. Nobody wrote them in `String`. Slide 13 shows the same chain for `String` and `Scanner`, and `Truck` reaching `Object` through `Vehicle`.
4. **Run `InheritedBehavior`.** `getClass()` is a behavior from `Object` that reports which class an object belongs to. It prints `class java.lang.String` twice. Both objects can use it because every `String` is also an `Object`. Tell students `getClass` is not on the AP exam; it is used once here to show an inherited behavior working.
5. **A three-level chain** (slide 14). `Item` names no superclass; `Book` and `DVD` extend `Item`. Pairs answer the three questions. Expected: a `DVD` object has a title, a due date, and a run time, can check out, and has the behaviors from `Object`; a `Book` object has no run time, because `Book` and `DVD` are siblings; the superclass of `Item` is `Object`.

MISCONCEPTION `object` and `Object` are the same word

Lowercase **object** is the general word for any instance of any class. Capitalized `Object` is the name of one class, the one at the top of every hierarchy. Java is case-sensitive (Topic 1.1).

Read "Every `String` is an object, and `String` is a subclass of `Object`" aloud and ask what each word refers to.

MISCONCEPTION Only classes that name a superclass have one

Students think `Vehicle` or `Person` has no superclass because none was written. Every class has one; if none is named, it is `Object`. And a class that does name a superclass is not cut off from `Object`: the chain keeps going up.

Day 1 independent practice

Hand out the puzzle worksheet. Students work Part B questions 4–6: a yes/no table for the streaming app's class hierarchy, two truths and a lie about the chain up to `Object`, and a face-off between the `String` class and the object "Ada". Whatever is not finished is homework. Circulate and check question 4: students who give a `Podcast` object an artist are reading sideways from Song, and students who give a `MediaItem` object a host are reading backward.

TEACHER NOTE

Questions 4–6 need only Day 1 material. Students keep the worksheet; the rest of it (questions 1–3 and 7–9) is Day 2 homework.

Day 1 closure

Point at four things on the board and have students call out "class" or "object" for each: `String` (class), "Hawks" (object), `Vehicle` (class), `Object` (class, the one at the top). Ask one student to explain why `Object` is a class even though its name is the word "object."

Day 2 warm-up: what is in your contacts list?

Ask: "Your phone's contacts list has your best friend in it. Is your friend inside your phone?" Students laugh and say no: the phone holds a number that lets you reach them. Then: "If you copy that number to a second phone, how many friends are there?" One. Two phones can reach the same person.

Then one Day 1 recall question, called out: "Is `String` a class or an object? Name an object of it." Class; any string literal.

TEACHER NOTE

Keep the analogy on the board all period. Every rule today maps onto it: the variable is the contacts entry, the reference is the number, and the object is the person. Copying an entry copies the number, not the person.

Day 2 direct instruction: reference types and object references

1. **Recall Topic 1.2.** Types are primitive (`int`, `double`, `boolean`) or reference. A primitive variable holds its value directly: `int score = 95;` puts 95 in the box named `score`.
2. **The problem.** A `String` can hold three letters or a whole novel. A variable's box cannot grow to fit any object.
3. **Reference types** (1.12.B.1, slides 15–16). A type named by a class, such as `String` or `Scanner`, is a **reference type**. A variable of a reference type holds an **object reference**, which can be thought of as the **memory address** of the object. The object itself is stored elsewhere.
4. **Run PrimitiveAndReference** (slide 16 afterward). Draw the memory picture: `score`'s box holds 95; `name`'s box holds a made-up address such as `#5021`, with an arrow to a separate box labeled `"Ada"`. Real addresses are chosen by Java; you never write them.
5. **Why it prints Ada.** `System.out.println` follows the reference and displays the object's text. The address stays hidden, which is convenient but can fool students into thinking `name` contains the letters.
6. **Change one thing.** Replace `"Ada"` with a 300-page novel. `name`'s box does not grow; only the object at the other end is bigger. Slide 16 shows the same picture with `seats` and `team`.

TEACHER NOTE

Build the memory picture with sticky notes: one color for variables (write the name and what the box holds), the other for objects (write the text). Students physically move the arrow or the address in the next section.

MISCONCEPTION The variable contains the text

Because `println(name)` shows `Ada`, students think the box holds `A, d, a`. It holds a reference; `println` follows it. The phone analogy helps: calling a contact reaches your friend, but the contacts list still holds only a number.

Day 2 direct instruction: declarations whose type must match

1. **Declaring a reference variable** (1.12.B.1). Use the class name as the type:
`String name = "Ada";`. Walk through the pieces: `String` (a reference type, capitalized), `name` (the variable), `=` (assignment, Topic 1.4), `"Ada"` (a string literal, Topic 1.3, which gives Java a `String` object; `name` gets a reference to it).
2. **The type must match.** A `String` variable can refer only to a `String` object; an `int` variable can hold only an `int`. The compiler checks this.
3. **Predict: WrongType.** Students find both errors before you compile. Line 5 writes `string` with a lowercase `s`: cannot find symbol (symbol: class `string`), because there is no class named `string`. Line 6 puts a `String` into an `int`:
incompatible types: `String` cannot be converted to `int`. Both are compiler errors, so nothing runs.

MISCONCEPTION string works like int

Primitive type names are lowercase, so students write `string`. Class names are capitalized, and Java is case-sensitive. Tie it to Topic 1.2's visual clue: lowercase `int`, `double`, `boolean` are primitive; a capitalized type name such as `String` or `Scanner` is a class, so it is a reference type.

Day 2 direct instruction: reassigning and copying references

1. **The rules from Topic 1.4 still hold.** Assigning replaces the value in a box; `a = b`; copies the value of `b` into `a`. For reference variables, the value in the box is a reference.
2. **Predict: CopyReference** (slide 17 afterward). Students write the two lines of output, then trace with sticky notes and made-up addresses. Line 1: `name` holds #5021 ($\rightarrow$ "Ada"). Line 2: `backup = name` copies #5021; two variables, one object. Line 3: `name` gets #7340 ($\rightarrow$ "Grace"); only `name`'s box changes. Output: Grace, Ada. Slide 17 shows the same pattern with `home` and `trip` as a trace table.
3. **Change one thing: CopyReferenceSwapped.** The same statements with lines 2 and 3 swapped: `name` is reassigned before the copy, so `backup` copies #7340, and the output is Grace twice. The copy takes whatever reference the variable holds at that moment (sequencing, Topic 1.1).
4. **Nothing changed the "Ada" object.** Reassigning a variable points it at a different object; the old object is untouched.

MISCONCEPTION After a copy, the two variables are linked

Students who learn that two variables can refer to the same object conclude that changing one changes the other. Each variable has its own box (slide 18). `backup = name` copied a reference once; reassigning `name` afterward replaces only `name`'s reference. Insist on a trace table with one column per variable, updated line by line.

Day 2 guided practice: track every box and the playlist bug

Start with slide 19 (Track every box). Pairs fill in the trace table, then write the output. Expected: after line 3, `x`, `y`, `z` refer to "cat", "dog", "cat"; after line 4, "dog", "dog", "cat"; after line 5, "dog", "emu", "cat". Output: dog, emu, cat. Students who print `emu` for `x` treated `x = y` as a lasting link.

Then show `PlaylistBug` in the IDE. The program should display the song that was playing before the change, then the song playing now (`Morning Light`, then `City Rain`), but it displays `City Rain` twice. Pairs trace it with a table, explain the error using references, and propose a fix. Debrief: line 3 copies the reference `nowPlaying` holds at that moment, which already leads to "City Rain". Nothing refers to "Morning Light" any more. Moving the copy above the reassignment fixes it: run `PlaylistFixed`. It is a logic error (Topic 1.1): the code compiles and runs, and only comparing the output to the expected outcome reveals it.

TEACHER NOTE

Students who finish early: ask what the output would be if line 3 were `String previous = "Morning Light";`. It would display the right songs, but only because the title was typed again: it no longer copies anything from `nowPlaying`, so it would be wrong for any other song.

Day 2 independent practice

Puzzle worksheet questions 1–3 and 7–9 are homework, about 20 minutes: a reference trace table, a be-the-JVM trace that ends in `length()`, a be-the-compiler table of declarations, code tiles that swap two references with a temporary variable, a one-token fix for a program that reassigns the wrong variable, and a who-am-I vocabulary review across both days. Question 1 settles the worksheet's Think First prediction.

Day 2 closure

Leave slide 20 (Words to know) up for a moment, then show slide 21. Pairs use the sentence starter: `String` is the **class**, `"Kai"` is one **instance** (object) of it, and `name` holds a **reference** to that object. Then ask the class to finish each sentence aloud: a reference variable holds **a reference to an object**; assigning to it **replaces its reference**; copying it **copies the reference, not the object**.

Day 2 exit ticket

Give the exit ticket (about five minutes). Question 1 checks the class/object relationship, a hierarchy, and `Object` (1.12.A); question 2 checks tracing reference variables through a copy and a reassignment (1.12.B). Re-teach the weaker one at the start of Topic 1.13.