

1.13 Object Creation and Storage

(Instantiation)

Lesson Plan

Pacing: 2 class periods of 45 minutes**Learning objectives:** 1.13.A, 1.13.B, 1.13.C**Essential knowledge:** 1.13.A.1, 1.13.A.2, 1.13.A.3, 1.13.B.1, 1.13.C.1, 1.13.C.2, 1.13.C.3, 1.13.C.4**Materials**

- Projector and the Under Construction slide deck
- Whiteboard and markers
- Colored pencils or highlighters for the Day 1 mark-the-text activity
- IDE on every computer
- Printed Under Construction puzzle worksheet (answer key for you) and exit ticket

Agenda

Minutes	Activity
5	Day 1 warm-up — the line you have typed since Topic 1.4 (slide 2)
10	Day 1 direct instruction — constructors and the new keyword (slides 3–7)
6	Day 1 guided practice — mark the text
10	Day 1 direct instruction — reference variables, null, and empty objects (slides 8–11)
7	Day 1 direct instruction — constructor signatures and starting values (slides 12–13)
5	Day 1 independent practice — puzzle worksheet questions 1, 2, 6, and 8
2	Day 1 closure — count the news
4	Day 2 warm-up — which String constructor?
7	Day 2 direct instruction — overloaded constructors (slide 14)
7	Day 2 direct instruction — arguments must match the parameter list (slides 16–17)
6	Day 2 guided practice — pick the constructor (slide 15) and the Backpack constructors
7	Day 2 direct instruction — call by value (slide 18)
6	Day 2 direct instruction — constructor calls and the flow of control (slides 18–19)
3	Day 2 closure — one statement, every idea (slides 20–21)
5	Day 2 exit ticket

Overview

Two periods on three learning objectives. Day 1 is about building and storing an object: an object is created with the keyword `new` followed by a call to one of the class's constructors, which have

the same name as the class (1.13.C.1, 1.13.A.1); the reference that comes back is stored in a variable whose type is the class, and a reference variable holds `null` when there is no object (1.13.B.1); a constructor's signature is its name plus the ordered list of parameter types, read from the header's parameter list (1.13.A.2); and the values passed in set the new object's starting values (1.13.C.2). Day 2 is about choosing and running a constructor: overloaded constructors have different signatures (1.13.A.3); arguments must be compatible with the parameter list in number, order, and type, and they are passed by value (1.13.C.3); and a constructor call jumps into the constructor and back (1.13.C.4).

The CED's suggested skill is 2.B (write program code involving data abstractions). The guided practice and puzzle worksheet ask students to build and correct complete constructor calls, not just recognize them.

Stay inside the topic boundary:

- `String` and `Scanner` are the running examples. Students already use `Scanner console = new Scanner(System.in);` (Topic 1.4) and `nextInt`, `nextDouble`, and `nextLine` to read input. Calls such as `reader.nextInt()` appear as in Topic 1.4; say once that the rules for calling a method on an object are taught fully in Topic 1.14.
- Constructor code inside a class is read, never written. `Ticket`, `Locker`, `Pizza`, `Backpack`, `Lamp`, `Parcel`, `Sprout`, `Kite`, and `Camera` are invented for this lesson so students can see inside a constructor; writing classes and constructors is Unit 3.
- A string literal such as `"Ada"` still gives a `String` object without `new`. That shortcut belongs to `String` alone and is covered in Topic 1.15. Joining text with `+` is also Topic 1.15, so a label and a value still take a `print` followed by a `println`.
- Method headers for `main` start with `public static`. Students can read past them; they are explained in a later topic.

TEACHER NOTE

The slides are the Under Construction deck (21 slides, with speaker notes); slide numbers below refer to it. Day 1 uses slides 1–13 and Day 2 uses slides 14–21. The Under Construction puzzle worksheet (10 questions, about 30 minutes) covers both days: questions 1, 2, 6, and 8 need only Day 1 material, and questions 3–5, 7, 9, and 10 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, constructors and `new`:** `NewGreeting`, `KeyboardScanner`, and `MissingNew`.
- **Day 1, reference variables and `null`:** `SharedReference`, `NullThenObject`, `WrongVariableType`, `NullPrimitive`, and `EmptyVersusNull`.
- **Day 1, signatures and starting values:** `ScannerFromText`.
- **Day 2, overloaded constructors:** `TwoScanners` and `StringFromNumber`.
- **Day 2, arguments must match:** `Ticket`, with `TicketOrder`, `TicketMissingPrice`, and `TicketLossy` for the three broken calls.
- **Day 2 guided practice:** `BackpackCalls` and `BackpackNoMatch`.
- **Day 2, call by value and flow of control:** `Locker` and `Pizza`; the study guide's `SeatCopy`, `SeatExpression`, and `ConstructorFlow` if students want a second look.
- **Printed only (exit ticket):** `ExitPets`, and `CameraCalls` with `CameraNoMatch` for the key.

To keep each demo in one file, `Ticket`, `Locker`, and `Pizza` put `main` inside the class whose constructor they call. The study guide's versions put the invented class in its own block below `main`; both run the same way. The deck invents its own small classes: the `Pizza` on slide 15 (sizes and toppings) is not the `Pizza` demo, and the `Ticket` on slides 18–19 has a single `int` parameter. Compiler messages are quoted from a recent JDK; the wording can vary slightly between Java versions.

Day 1 warm-up: the line you have typed since Topic 1.4

Show slide 2, or write `Scanner console = new Scanner(System.in);` on the board. Pairs answer three questions in two minutes: "Which word is the name of a class?", "After this line runs, does an object exist? What kind?", and "What does `console` hold: the object, or something else?"

Expected: `Scanner` is the class (it appears twice); yes, one `Scanner` object exists; and from Topic 1.12, `console` holds a reference to the object, which you can picture as its address. Then ask the question nobody can answer yet: "What does the word `new` do?" Leave it on the board; it is today's first idea.

TEACHER NOTE

Students who say "`console` is the Scanner" are reaching for the Topic 1.12 misconception that the variable contains the object. Ask them to say it again with the word "refers": `console` refers to a `Scanner` object.

Day 1 direct instruction: constructors and the new keyword

1. **The construction crew** (slides 3–5). Read the five goals on slide 3; the first three are today's. A blueprint is not a house; someone has to build the house. Every class comes with special code whose only job is to build objects of that class: a **constructor** (1.13.A.1). A constructor has the **same name as its class**, capital letter and all.
2. **new** (1.13.C.1). An object is created by writing the keyword `new` followed by a call to one of the class's constructors. `new` means "build a brand-new object"; the constructor call after it says which class and with what information. Creating an object is called **instantiation**: the result is a new instance (Topic 1.12).
3. **Take one statement apart** (slide 5). `String greeting = new String("Good morning");` Piece by piece: `String greeting` declares a variable that can refer to a `String`; `=` stores the right side's result (Topic 1.4); `new` asks for a new object; `String(...)` is the constructor call; "Good morning" is the information passed in. Java runs the right side first, and the result of `new` is a reference to the new object, which `=` stores in `greeting`. (`NewGreeting` prints `Good morning`.)
4. **Same shape, another class** (slide 6). Run `KeyboardScanner`. Line 7 has exactly the same shape: `new Scanner(System.in)` calls the `Scanner` constructor with `System.in` (the keyboard), and the reference is stored in `keyboard`. It prints `A Scanner object is ready.` and nothing needs to be

typed: building a `Scanner` only sets it up to read. Slide 6 lines up three statements to show the pattern `Type name = new Type(...);`.

5. **Predict: MissingNew** (slide 7 afterward). Without `new`, does it compile? No: Java reports cannot find symbol and names method `String(String)`. Without `new`, Java thinks this is an ordinary method call (Topic 1.9) to a method named `String`, and there is none.
6. **The literal shortcut**. Students will ask why `String name = "Ada";` works without `new`. A string literal is a shortcut only `String` gets; Topic 1.15 covers it. Every other class's objects come from `new`.

MISCONCEPTION Leaving out new

A constructor call looks like a method call from Topic 1.9, so students write `Scanner input = Scanner(System.in);`. The compiler's own message is the best teacher: it says it cannot find a **method** named `Scanner`. Have students read the message aloud and then say the rule: `new` is what tells Java this is a request to build an object.

Day 1 guided practice: mark the text

This is the CED's "Marking the Text" strategy. Write the four statements on the board. Each student copies the four statements and marks them with colors: **circle** the assignment operator and the keyword `new`; **underline** the variable's type and the constructor being called; **draw a box** around the list of arguments passed to the constructor.

- `Scanner keyboard = new Scanner(System.in);`
- `String city = new String("Denver");`
- `Ticket mine = new Ticket("Dune", 14, 12.5);`
- `String empty = new String();`

Debrief by asking: "In every statement, what do the two underlined words have in common?" They match: the variable's type is the class being built. "What is in the box in the last statement?" Nothing: that constructor takes no arguments. "How many objects exist after all four run?" Four, one per `new`.

TEACHER NOTE

`Ticket` is an invented class (a movie ticket) used all through the topic. Tell students they will see inside its constructor on Day 2, and that they will write classes like it in Unit 3. Keep the marked statements; the Day 1 closure and the Day 2 warm-up return to them.

Day 1 direct instruction: reference variables, null, and empty objects

1. **Where does the reference go?** (slide 8). If the reference `new` hands back is not stored, the program can never find the object again. It goes in a variable whose **type is the class**: `Scanner keyboard;`, `String city;`. It looks like a Topic 1.2 declaration with a class name where a primitive type would go.

2. **Predict: SharedReference** (slide 9 afterward). How many objects? Line 2 has no `new`, so it builds nothing; `=` copies the reference already in `city` (Topic 1.12). Both lines print `Denver`, and there is **one** object. Rule of thumb: count the `news` to count the objects.
3. `null` (1.13.B.1). A variable of a reference type holds an object reference or, if there is no object, the literal `null` (Topic 1.4). Run `NullThenObject`: it prints `null`, then `Sky`. Line 3 creates an object and its reference replaces `null`, like any assignment.
4. **Empty is not null** (slide 10). Run `EmptyVersusNull`. `new String()` builds a real object with no characters, so the brackets print with nothing between them: `[]`. The `null` variable prints `[null]`, because there is no object at all.
5. **Predict: which declarations compile?** (slide 11). Pairs decide for the four declarations. Expected: A (`String title = null;`) and C (`Scanner in = new Scanner(System.in);`) compile; B (`double price = null;`) and D (`String reader = new Scanner(System.in);`) do not. Then compile the two demo files for the messages: `WrongVariableType`:
incompatible types: Scanner cannot be converted to String. `NullPrimitive`:
incompatible types: <null> cannot be converted to int. A `String` variable holds only references to `String` objects; an `int` always holds a number, never a reference, so it cannot hold `null`.

MISCONCEPTION Any variable can hold any object

Students copy a declaration they know, such as `String`, and put a different object in it. The fix comes from the marked statements: underline the class after `new`, then make the variable's type the same word. The compiler checks this, which is why `WrongVariableType` does not compile.

MISCONCEPTION null works for every type

Students who learn that `null` means "nothing" try `int count = null;` to mean "no count yet." `null` means "no object," so it fits only reference types. A primitive variable always holds a value of its type.

MISCONCEPTION An empty object is the same as null

Both seem to mean "nothing," so students mix them up. `new String()` is an object that exists and has no characters; `null` is the absence of an object, and no constructor ran. The `EmptyVersusNull` output, `[]` beside `[null]`, makes the difference visible.

Day 1 direct instruction: constructor signatures and starting values

1. **Reading a header** (slides 12–13). How do you know what goes in the parentheses for a class you have never used? Read the constructor's header in the documentation, as in Topic 1.9. Two real ones: `String(String original)` and `Scanner(InputStream source)`. A constructor header has the class name and **no return type**, not even `void`.
2. **Parameter list and signature** (1.13.A.2). The **parameter list** gives the type and a variable name for each value the constructor needs. The **signature** is the constructor's name plus the

ordered list of parameter types; the names are left out. `String(String original)` has the signature `String(String)`; `String()` has the signature `String()`.

3. **A made-up header** (slide 13). `Ticket(String movie, int seat, double price)` has the signature `Ticket(String, int, double)`. A call needs three values in that order: a `String`, an `int`, then a `double`. Order is part of the signature: `Ticket(int, String, double)` would be a different one.
4. **Parameters set starting values** (1.13.C.2). A new object has to start out holding something, and the constructor cannot guess. The values passed in set the starting values of the new object's attributes. Run `ScannerFromText: new Scanner("85 92")` builds a `Scanner` whose starting text is `85 92` (the `reader` line on slide 6), so the program prints `85` and `92` with no typing. Change the argument and the same lines read different numbers.

MISCONCEPTION Parameter names are part of the signature

Students see `Ticket(String movie, int seat, double price)` and `Ticket(String title, int row, double cost)` and call them different. Drop the names: both are `Ticket(String, int, double)`. The study guide's `DuplicateSignature` shows that a class cannot contain both: the compiler says the constructor `Ticket(String,int,double)` is already defined, describing it by its signature, with no names.

Day 1 independent practice

Hand out the puzzle worksheet. Students work questions 1, 2, 6, and 8: counting the objects each statement creates, a be-the-compiler table of declarations and constructor calls, a face-off between an empty `String` and `null`, and a one-name fix for a variable that copied `null` before the object existed. Whatever is not finished is homework. Circulate and check question 1: students who count an object for `String b = a;` or `String c = null;` are not yet counting the `news`.

TEACHER NOTE

Questions 1, 2, 6, and 8 need only Day 1 material. Students keep the worksheet; the rest of it (questions 3–5, 7, 9, and 10) is Day 2 homework.

Day 1 closure: count the news

Point at the four marked statements from guided practice. Ask: "How many objects do these four statements create?" (Four.) Then add `String sameCity = city;` and ask again. (Still four: no `new`, no new object.) One student finishes the sentence: "A constructor runs only when ..." (`new` calls it).

Day 2 warm-up: which String constructor?

Show `new String()` and `new String("Lake")`. Ask: "Both call a constructor named `String`. How many arguments does each pass? What does each new object hold?" Expected: zero arguments,

and the object holds no characters; one `String` argument, and the object holds `Lake`. Then: "So how many `String` constructors are there?" At least two, which is today's first idea.

Day 2 direct instruction: overloaded constructors

1. **Why more than one?** People want objects that start in different ways: a `String` with given characters or an empty one; a `Scanner` that reads the keyboard or one that reads text you already have.
2. **Overloaded constructors** (1.13.A.3, slide 14). A class's constructors all share the class name, so they are told apart by their parameter lists. Constructors are **overloaded** when a class has more than one constructor with different signatures: a different number of parameters, different types, or the same types in a different order. The same idea as overloaded methods in Topic 1.9.
3. **How Java chooses** (1.13.A, slide 14). It matches the call's arguments against each signature and runs the one that fits: slide 14's `Lamp` calls each pick a different constructor by their arguments. That is also why two constructors may never share a signature: Java could not choose.
4. **Predict: TwoScanners**. Both calls pass one argument, so the **type** decides: `System.in` is an `InputStream`, so line 7 runs `Scanner(InputStream)`; `"70 88"` is a `String`, so line 8 runs `Scanner(String)`. It prints 70.
5. **Predict: StringFromNumber**. `new String(5)` does not compile:
no suitable constructor found for `String(int)`. Overloading gives a class only the constructors that were written, and no `String` constructor takes an `int`.

MISCONCEPTION There is a constructor for whatever you pass

Students who have seen several constructors start to assume one exists for any argument. Before calling a constructor, find a signature that matches the arguments. The compiler's message for `StringFromNumber` lists the `String` constructors it tried, including some this course never uses; the point is only that none of them takes an `int`.

Day 2 direct instruction: arguments must match the parameter list

1. **Arguments and parameters** (1.13.C.3, slide 17). An **argument** is a value passed in when the constructor is called. Java pairs arguments with parameters **by position**, so three things must line up: the **number** of arguments, their **order**, and their **types**. Each argument must be **compatible** with its parameter: that type, or one Java converts to automatically. From Topic 1.5, an `int` converts to a `double` automatically; a `double` never converts to an `int` on its own.
2. **Run Ticket**. `new Ticket("Dune", 14, 12)` prints Dune, 14, and 12.0. The third argument is the `int` 12 going to `double` `price`; it is compatible, and the output shows the conversion.
3. **Change one thing**. Three calls, each one small change from a call that works, and none compiles (slide 17 shows them beside the call that works):

Call	What is wrong	Compiler message (example file)
------	---------------	---------------------------------

<code>new Ticket(14, "Dune", 12.5)</code>	Order	incompatible types: int cannot be converted to String (TicketOrder)
<code>new Ticket("Up", 4)</code>	Number	actual and formal argument lists differ in length (TicketMissingPrice)
<code>new Ticket("Dune", 12.5, 14)</code>	Type	incompatible types: possible lossy conversion from double to int (TicketLossy)

In the last call, 14 going to `double price` would be fine; the problem is 12.5 going to `int seat`.

MISCONCEPTION Ordering arguments by what makes sense

Students write values in the order they think of them ("seat 14 for Dune") instead of the order of the parameter list. Java does not know that 14 "looks like" a seat; it only knows the second parameter is an `int`. Have students write the header above the call and draw a line from each argument to its parameter, left to right.

Day 2 guided practice: pick the constructor and the Backpack constructors

Go back to slide 15 (Pick the constructor). Pairs decide which `Pizza` constructor each call runs, or "none." Expected: A runs `Pizza(String, int)`; B does not compile, because 2 would go to `String size`; C does not compile, because this class has no constructor with no parameters; D runs `Pizza(String)`. Call out C: a class has only the constructors that were written.

Then the harder round on the board. An invented class `Backpack` has four constructors: `Backpack()`, `Backpack(String color)`, `Backpack(String color, int pockets)`, and `Backpack(int pockets, double weight)`. Pairs decide, for each of six calls, which constructor runs, or that no constructor matches and the call does not compile.

Call	Answer
<code>new Backpack()</code>	<code>Backpack()</code>
<code>new Backpack("green", 3)</code>	<code>Backpack(String color, int pockets)</code>
<code>new Backpack(3, 2)</code>	<code>Backpack(int pockets, double weight)</code> ; the int 2 becomes 2.0
<code>new Backpack(3, "green")</code>	No match: no constructor takes an <code>int</code> followed by a <code>String</code>
<code>new Backpack(2.5, 3)</code>	No match: the only constructor that starts with a number needs an <code>int</code> first, and 2.5 is a <code>double</code>
<code>new Backpack("green")</code>	<code>Backpack(String color)</code>

Debrief by running `BackpackCalls`, whose constructors print which one ran: `Backpack()`, `Backpack(String, int)`, `Backpack(int, double)` with weight 2.0, and `Backpack(String)`. Then compile `BackpackNoMatch` to see no suitable constructor found for `Backpack(int, String)` and no suitable constructor found for `Backpack(double, int)`. Notice that the compiler describes each failed call by its argument types, the same form as a signature.

TEACHER NOTE

Early finishers: "Could the class also have `Backpack(String name, int count)`?" No: its signature is `Backpack(String, int)`, the same as an existing constructor, so the class would not compile.

Day 2 direct instruction: call by value

1. **The question.** "If a constructor changes its parameter, does your variable change too?" If it could, you could never trust a variable after a call.
2. **The rule** (1.13.C.3, slide 18). Arguments are passed using **call by value**: each parameter starts as a **copy** of its argument's value. Like photocopying a form before handing it over: whatever is written on the copy, your original stays as it was.
3. **Predict: Locker.** The constructor adds 100 to its parameter. Students predict both lines. Output: 112, then 12. Trace with two columns: `myNumber` in `main` stays 12 the whole time; `number` starts as a copy of 12, becomes 112, and is gone when the constructor finishes. Slide 18 shows the same copy with `mySeat` staying 14.
4. **Arguments can be expressions.** Java evaluates an argument first and passes a copy of the result. The study guide's `SeatExpression` passes `row * 10`, so `seat` starts at 30 and `row` is still 3.

MISCONCEPTION The parameter is the same variable as the argument

Students expect `myNumber` to become 112 "because it was passed in." The constructor never sees `myNumber`, only a copy of its value. Even if both variables had the same name, they would be separate variables. Puzzle worksheet question 4 checks the same idea with `water` and `cups`.

Day 2 direct instruction: constructor calls and the flow of control

1. **The rule** (1.13.C.4, slide 18). A constructor call interrupts the normal top-to-bottom order. Java runs every statement in the constructor first; once the last one finishes, control returns to the point right after the call, finishes that statement (storing the reference), and moves on. The order statements actually run in is the **flow of control**.
2. **The analogy.** Pausing a video to answer the door: you deal with the visitor completely, then resume exactly where you stopped.
3. **Predict: Pizza.** Students write the four lines of output. `Open`; line 6 jumps into the constructor, which prints `cheese`; back to line 7, `Next customer`; line 8 jumps in again and prints `veggie`. Each `new` makes its own jump and returns to just after its own call.
4. **Check: slide 19** (Line up at the movies). Pairs write the output: `Line up`, `Ticket 5`, `Buy popcorn`, `Ticket 6`, `Find seats`.

MISCONCEPTION The file order is the run order

The constructor sits below `main` in the file, so students predict its output comes last, or never. Code in a constructor runs only when a call reaches it, and it runs right then. Trace by following the calls, not the line numbers on the page. The study guide's `ConstructorFlow` labels its output A, B, C, D to make the order visible.

Day 2 independent practice

Puzzle worksheet questions 3–5, 7, 9, and 10 are homework, about 20 minutes: a snippet pool that picks among overloaded `Parcel` constructors, a be-the-JVM trace of two constructor calls and call by value, code tiles that read a city and a box count with `Scanner` and build a `Parcel`, two truths and a lie about `Kite` signatures, a who-am-I vocabulary review, and a one-sentence reflection on how Java chooses a constructor. Question 4 settles the Think First prediction printed just before it.

Day 2 closure: one statement, every idea

Leave slide 20 (Words to know) up. Write `Ticket mine = new Ticket("Up", seat, 9);` on the board, with `int seat = 5;` above it. Ask the class to name, in order: the variable's type, the keyword that creates the object, the signature that must exist, the conversion that happens (the `int 9` becomes `9.0`), what `seat` holds after the call (still 5: call by value), and when the next line runs (after every statement in the constructor). If time allows, slide 21's sentence starter takes the warm-up line apart one last time as a pair-share.

Day 2 exit ticket

Give the exit ticket (about five minutes). Question 1 checks choosing a constructor by its signature and matching arguments in number, order, and type (1.13.A, 1.13.C); question 2 checks reference variables, `null`, and counting the objects `new` creates (1.13.B, 1.13.C). Re-teach the weaker one at the start of Topic 1.14.