lectures # week-5

Week 5: Inheritance in the Memory Model & Static vs Dynamic Binding (Wednesday, February 4, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A2 and completes the memory model exercise from Week 5 Day 1.

Lecture Preview [0:19]

Today's agenda:

1. **Assignment 2 Q&A** — Enum null testing, `chooseDirection` vs `canPass`, helper methods
2. **Memory model completion** — Finish the B object construction trace, garbage collection
3. **Compile-time vs runtime binding** — How the compiler and JVM resolve method calls differently
4. **Casting** — Telling the compiler "trust me" (and why you shouldn't)

Coming Friday

More on polymorphism (the `advance` method in Road Rage), then event-driven programming and GUI development begins.

Assignment 2 Q&A [1:21]

Enum Null Testing [1:25]

Student Question

Q: For enums, do we need to test for null?

Enums are singletons. Each enum constant (e.g., `Direction.NORTH`) is a single object—there is one and always only one of each. You can test for completeness across all enum values.

But a **variable** of an enum type can hold one of the enum values *or* null:

```
Direction currentDirection = Direction.NORTH;  // one of four values
currentDirection = null;                        // also valid
```

So `currentDirection` has five possible values: the four directions, plus null.

Similarly, a method parameter of enum type can receive null:

```
public boolean canPass(final Terrain theTerrain, final Light theLight) {
    // theTerrain could be null - compiler won't catch it
}
```

The compiler will reject invalid types (you can't pass an `int` or `String`), but it will **not** check for null.

The answer: No, you don't need to check for null. The specification does not require defensive coding against null. You won't lose points for omitting the check.

But If You Do Add a Null Check...

By adding an explicit null check (e.g., `Objects.requireNonNull()`), you've **changed the spec**. You've added behavior beyond what was documented. That means:

1. You must **document it** — update your Javadoc so testers know
2. You must **test it** — write tests for that new behavior

Adding a defensive check without documentation is undocumented behavior.

Specs vs Extra Features [9:16]

The scenario: You have a deadline. Stakeholders are there for the demo. The spec says do A, B, and C. You demo A, B, and this incredible extra feature Z—maybe even D from next month's sprint.

The first question asked: *"Where is C?"*

The takeaway: complete the specification first. Don't add features that aren't required.

That said: if you notice a deficiency in the specification as a junior developer, speak up. Go to a senior dev or project manager:

- *"What does this requirement mean?"*
- *"I can't code this as written—it doesn't make sense."*
- *"We should probably do some defensive coding here—the system might evolve."*

Worst case, they say stop wasting their time. Best case, they see you're thinking beyond the minimum. Either way, you're not wrong for raising it.

`chooseDirection` vs `canPass` [14:22]

These two methods feel related—in the simulation loop, they're called one after the other. But **implement them as standalone methods**:

Method	Input	Responsibility
`chooseDirection`	Map of neighbor terrains	Given my neighbors, which direction do I choose based on my rule structure?
`canPass`	One terrain + light status	Given this terrain and light, can I go forward?

Do not call `canPass` from inside `chooseDirection`. The parameters don't align—`chooseDirection` receives a map of terrains (no light status), while `canPass` requires a terrain and a light. You could hack it by passing a green light, but that's a trick—not good design.

The result is that some vehicles (like the bicycle) will choose a direction toward a traffic light, then have to stop and wait because the light is red. That's correct behavior—the two methods operate independently.

Helper Method: `isValidTerrain` [22:13]

A common approach in `chooseDirection` is a big boolean expression:

```
if (map.get(Direction.NORTH) == Terrain.GRASS
        || map.get(Direction.NORTH) == Terrain.STREET
        || map.get(Direction.NORTH) == Terrain.LIGHT /* ... */) {
    // go north
} else if (map.get(Direction.LEFT) == Terrain.GRASS
        || map.get(Direction.LEFT) == Terrain.STREET /* ... */) {
    // go left
}
```

This is **fragile code**. If the specs change—a new terrain is added, or a vehicle's valid terrains change—you have to update that boolean expression in three or more places.

Better approach: Extract a helper method:

```
private boolean isValidTerrain(final Terrain theTerrain) {
    return theTerrain == Terrain.GRASS
            || theTerrain == Terrain.STREET
            || theTerrain == Terrain.LIGHT;
}
```

Now the if-statement reads contextually:

```
if (isValidTerrain(map.get(Direction.NORTH))) {
    // go north
} else if (isValidTerrain(map.get(Direction.LEFT))) {
    // go left
} else if (isValidTerrain(map.get(Direction.RIGHT))) {
    // go right
}
```

ATV example: The ATV can go anywhere except walls. Its helper is one line:

```
private boolean isValidTerrain(final Terrain theTerrain) {
    return theTerrain != Terrain.WALL;
}
```

Even though it's a tiny expression, the pattern is worth keeping—it gives the boolean a contextual name and centralizes the terrain logic.

Thinking About Robustness

Ask yourself: *"If ten new terrains were added, how much code would I have to change across all my classes?"* If the answer is "just one place per class," you're in good shape.

Alternative approach: A constant `Set` of valid terrains works well too. If terrains change, you just update the set contents.

Don't Over-Centralize

Don't try to put all vehicle terrain rules in the abstract parent class. Each vehicle type should manage its own valid terrains—it's the car's responsibility to know the car's valid terrains, not the abstract vehicle's.

Tracking Disabled State [30:53]

Each vehicle type has a maximum disabled duration (e.g., truck = 5, car = 25, bicycle = 45). Every individual vehicle object needs to track how long it's been disabled.

Do not store a boolean for `isDisabled`. Calculate whether you're disabled based on your tracking state. The `isDisabled()` method should compute its answer, not read a flag.

Taxi Hint

The taxi has one extra piece of **state** that it needs to keep track of—how long it has been sitting at a stoplight. In OOP, state is represented by **instance fields**. The specification doesn't restrict what instance fields you can add.

Truck Reversing [37:33]

You may see a truck appear to reverse direction on certain maps. It's not actually reversing—it's within the rule structure. For example:

1. Truck approaches a red light, blows through it (trucks ignore lights)
2. Truck randomly chooses to go toward a red crosswalk, turns and faces it, stops
3. Next turn, truck randomly chooses to go back the direction it came from

Your eye sees a reversal, but each individual move follows the truck's rules.

Memory Model: Completing the B Object [39:16]

Continuing from Monday's lecture, where we left off mid-construction of the B object.

Constructor Chaining: B → A → Object [40:18]

When `new B("Steven")` is called, the constructor chain builds stack frames:

1. **B's constructor** receives `#ABCB` (address of "Steven" in string pool)
2. **`super(theName)`** → A's constructor receives `#ABCB`
3. **`super()`** → Object's constructor (implicit)

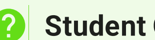
The call stack at this point (bottom to top):

Frame	Method
4	`Object()`
3	`A("Steven")`
2	`B("Steven")`
1	`memoryModelEx7()` — locals: `a`, `sString`
0	`main()`

Execution order (unwinding back down):

1. Object constructor runs → pops off
2. A constructor: `myName = #ABCB` ("Steven"), `myId = idGenerator++` → myId gets 2, idGenerator becomes 3 → pops off
3. B constructor: `myValue = theName.length()` → "Steven" is 6 characters → myValue = 6 → pops off
4. Returns `#671B` (address of new B object) to variable `a`

JDK 25: Statements Before `super()` [42:40]



Student Question

Q: Can I have statements before the call to `super()` in a constructor?

Before JDK 25: Statements before `super()` was a compiler error. Period.

JDK 25 and above: You **can** have statements before `super()`, but they **cannot access your own instance fields**.

```
public B(final String theName) {
    // JDK 25+: These statements are allowed before super()
    String processed = theName.trim().toLowerCase();  // ✓ OK

    super(processed);  // Now call parent constructor

    // After super(), you can use instance fields
    myValue = theName.length();
}
```

The restriction exists because instance fields aren't initialized until the parent constructor runs. So you can process parameters, but you can't touch `myValue` (or any inherited field) until after `super()`.

Note

If the compiler inserts an implicit `super()`, it always goes first—the compiler won't put it after your statements.

Metaspace Clarification [48:02]

A correction from the whiteboard: **Metaspace does not live in the heap.**

Memory Region	Managed By	Contents
Stack	JVM	Stack frames, local variables, method calls
Heap	JVM	Object instances, string pool
Metaspace	Native OS	Class objects, `static final` constants, class metadata

The class objects (`A.class`, `B.class`, `Object.class`, etc.) are stored in metaspace, which lives in the operating system's native memory—outside the JVM's managed heap. This gives even faster access.

For our level of understanding, this distinction doesn't change how the memory model works. But it's good to know that metaspace is separate from the heap.

Garbage Collection at the Breakpoint [51:04]

After `a = new B("Steven")`, we pause at a breakpoint. The variable `a` now references `#671B` (the B object). What happens if the garbage collector runs?

Still referenced (safe):

Address	Object	Referenced By
#671B	B object	`a` on the stack
#671A	"id: 1\nname: Charles"	`sString` on the stack
#ABCA	"Charles"	String pool
#ABCB	"Steven"	String pool
#ABCC	"\n****\n"	String pool
#9876	A.class	Metaspace
#9875	B.class	Metaspace

Orphaned (garbage collected):

Address	Object	Why
#6719	The original A object	`a` was reassigned to point to the B object—nothing references this A anymore

The A object at `#6719` gets marked for garbage collection. It's an orphan—no variable references it.

String Pool and Class Objects Persist

String pool objects stay for the life of the program (unless explicitly removed via an API call—not something we deal with). Class objects stay because metaspace maintains references to them.

Shared Mutable Objects [53:30]

Student Question

Q: In our example, both the A and B objects have `myName` fields that reference String objects in the string pool. Since Strings are immutable, neither object can change the string. But what if the fields referenced mutable objects instead—like if A had a `myPoint` field referencing a `Point` object, and both the A and B objects were given the same `Point`? Would changes through one object affect the other?

A: Yes—and this is where shared references to mutable objects get dangerous.

Two types of mutation can happen when objects share references:

Mutating the shared object: If both the A and B objects reference the same `Point`, and B changes the point's x-coordinate to 100—A sees that change too. They share the same object.

Reassigning the reference: If B calls a setter that assigns a *new* `Point` object, B now points to a different object. A still points to the original. They've diverged.

This is the distinction between mutating the **object** a reference points to versus mutating the **reference** itself.

B Object: Private Fields and Inheritance [57:16]

Look at the B object in memory. It has **three** instance fields:

Field	Declared In	Accessible from B?
`myName`	A (private)	No — but B *has* it

Field	Declared In	Accessible from B?
`myId`	A (private)	No — but B *has* it
`myValue`	B (private)	Yes

The B class only declares one field, but B objects inherit all of A's instance fields. The fields exist in the object's memory. They're allocated. They're *there*.

But B can't directly access them. They're private in A.

How does B access the data? Through A's **public methods**: `getName()`, `getId()`, `toString()`.

Private Fields vs Private Methods in Inheritance

These behave differently:

- **Private fields** in a parent → Child classes **have** them (allocated in memory) but **cannot access** them directly
- **Private methods** in a parent → Child classes **do not have** them at all

A private helper method in A is invisible to B. B doesn't know it exists. But if A has a public method that calls that private helper, B can call the public method—and the helper runs internally.

Compile-Time vs Runtime Binding [1:01:28]

This is the critical concept from today's lecture. Variable `a` is declared as type `A` but references a `B` object.

The Compiler's Perspective [1:01:37]

The compiler checks the **variable type**, not the object type. It has no idea what object the variable currently references.

Statement	Compiles?	Why
`a.getName()`	**Yes**	Compiler finds `getName()` in class A

Statement	Compiles?	Why
`a.convertName()`	**Yes**	Compiler finds `convertName()` in class A
`a.getValue()`	**No**	Compiler looks in class A—no `getValue()` method
`((B) a).getValue()`	**Yes**	Cast tells compiler: "treat this as type B"

The compiler looks at `a`, sees its type is `A`, and checks whether class A has the method being called. That's it.

Casting: "Trust Me, Compiler" [1:04:49]

When you write `((B) a).getValue()`, you're telling the compiler:

> "Thank you for your hard work keeping my code safe. But all of my programming experience has taught me that I know more than you. Trust me—this is a B object. Let me call this method."

The compiler says *"Okay, but if you get this wrong, you'll get a runtime exception."*

> ⚠ **Infrequently Cast**
>
> Casting should be infrequent. It bypasses the compiler's type safety—one of the most valuable protections against runtime errors. When we do cast later in the course (group project), we'll have checks in place to verify the cast is safe.

The Runtime's Perspective [1:09:00]

At runtime, the JVM knows the **actual object type**. It starts at the object's class and searches up the hierarchy for the method implementation.

Statement	Runtime Calls	Why
`a.getName()`	**A's** `getName()`	B doesn't override it—found in A

Statement	Runtime Calls	Why
`a.convertName()`	**B's** `convertName()`	B overrides it → `getName().toLowerCase()`
`((B) a).getValue()`	**B's** `getValue()`	Object is a B, method found in B

For `convertName()`: even though the compiler found `convertName()` in class A, at runtime the JVM sees the object is a B—and B overrides `convertName()`. So B's version runs, returning the name in lowercase instead of uppercase.

ClassCastException [1:12:39]

What if `a` were still referencing the original A object when we tried `((B) a).getValue()`?

The compiler would let it pass (we told it to trust us). But at runtime, the JVM would discover the object is an A, not a B. **ClassCastException**—program crashes.

This is exactly the risk of casting: you've told the compiler to skip its safety check, and you were wrong.

`@Override` Annotation [1:11:29]

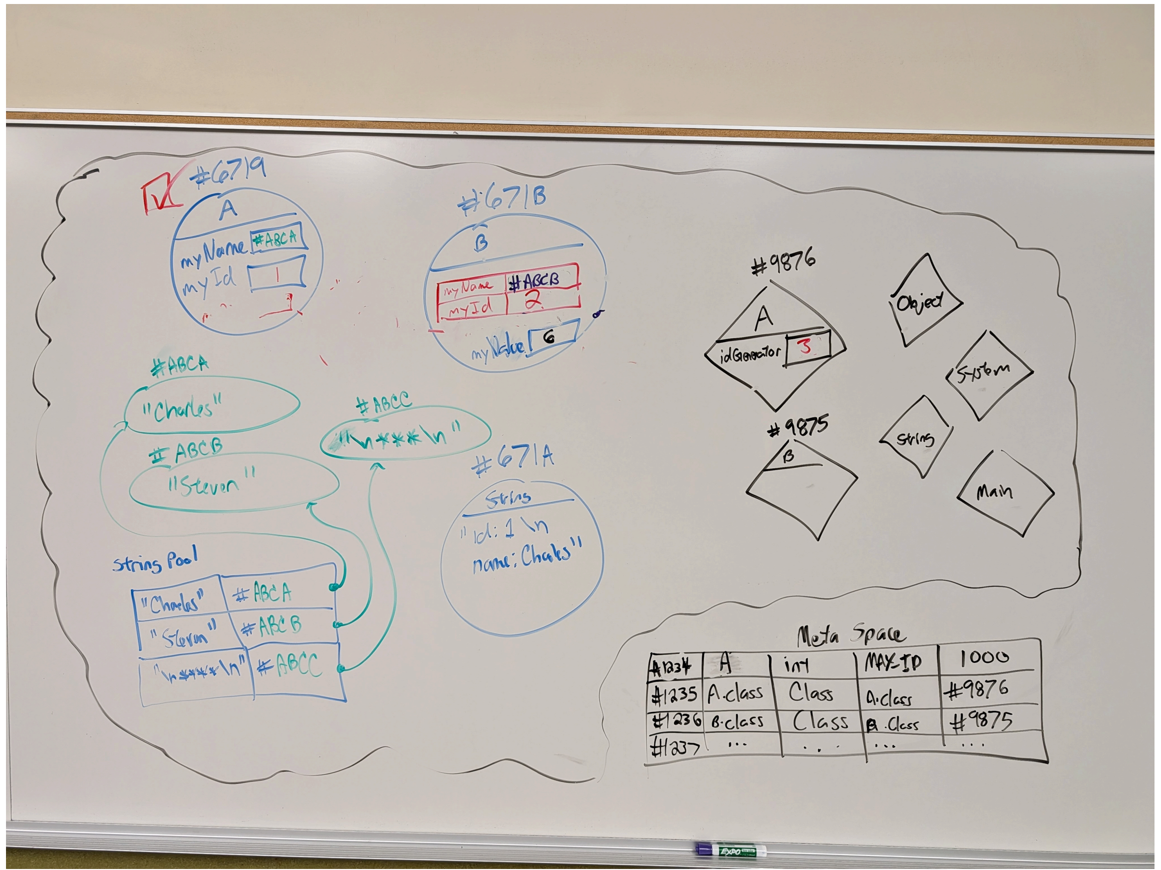
The `@Override` annotation tells the compiler: *"I intend to override this method—double-check that I actually am."*

If you accidentally overload (wrong parameter types) or have a typo in the method name, the compiler will catch it and give you an error. It's a safety net, not a functional requirement.

Whiteboard Photos

Complete Memory Model Diagram

The full heap diagram showing both A and B objects, string pool, class objects, and metaspace after completing the exercise.

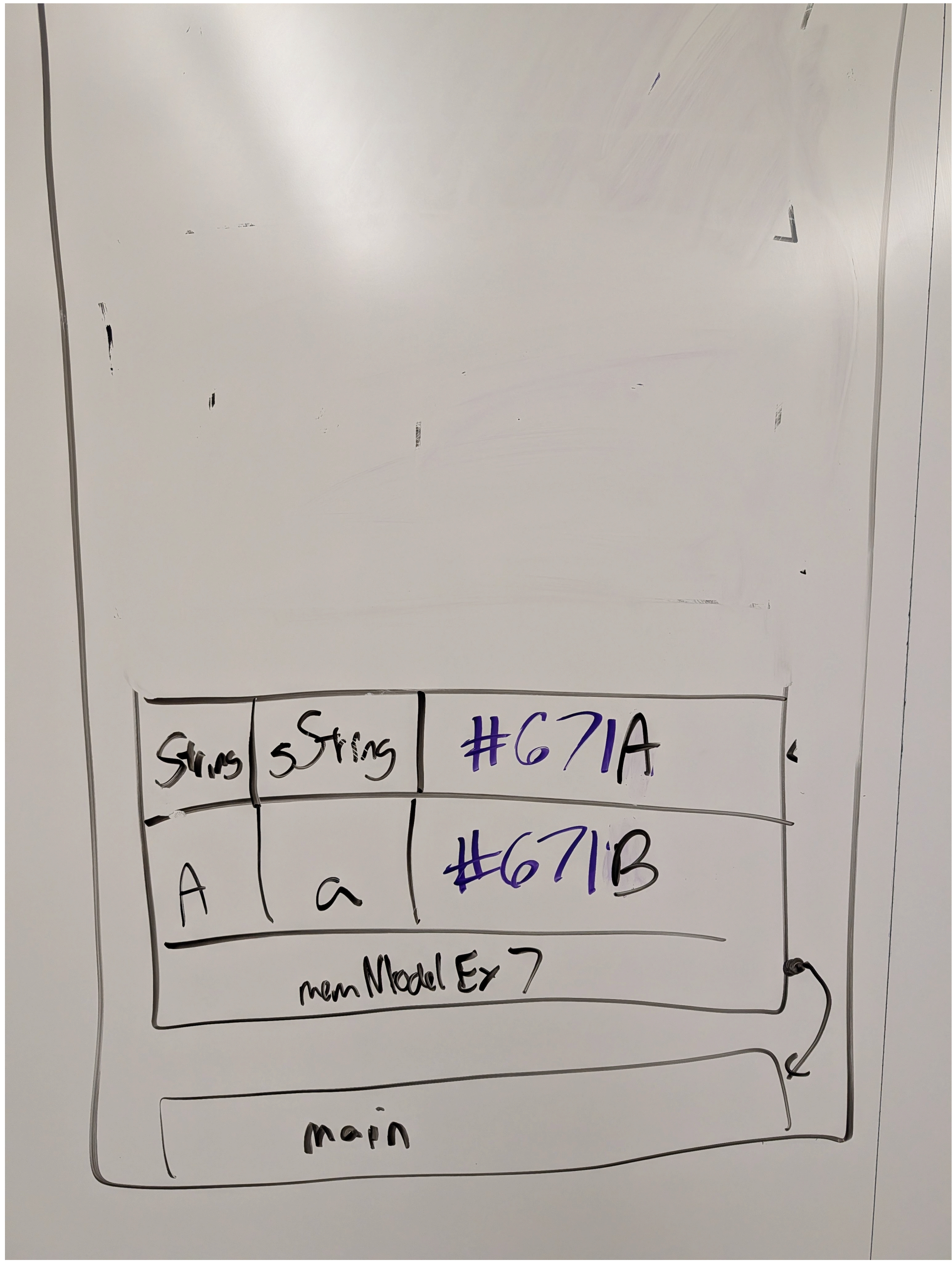


What's shown:

- **Heap** (left): A object at `#6719` (orphaned), B object at `#671B` with inherited fields in red, toString result at `#671A`
- **String Pool** (bottom left): "Charles" → `#ABCA`, "Steven" → `#ABCB`, "\n****\n" → `#ABCC`
- **Class Objects** (right): Diamond hierarchy—`Object`, `A` (with `idGenerator = 3`), `B`, plus `System`, `String`, `Main`
- **Metaspace** (bottom right): Table mapping class addresses, types, and constants (`A.MAX_ID = 1000`)

Stack Frame for `memoryModelEx7`

The call stack showing local variables at the breakpoint.



What's shown:

- `memModelEx7` **frame**: `String sString` → `#671A`, `A a` → `#671B`
- `main` **frame**: Below

Compile-Time vs Runtime Binding

The key diagram showing what the compiler allows vs what the runtime executes.

What's shown:

- **Compiler** (left): `a.getName()` ✓, `a.convertName()` ✓, `a.getValue()` ✗, `((B) a).getValue()` ✓ (with cast)
- **Runtime** (right): `getName()` → A class, `convertName()` → B class (overridden), `getValue()` → B class, but risks `ClassCastException` if object isn't actually a B

Lecture Demo Code

Same classes as Week 5 Day 1—the memory model exercise uses classes `A`, `B`, and `MemoryModelDemo`.

This lecture outline is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.