lectures # week-5

Week 5: Memory Model Continued (Monday, February 2, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A2 and continues the memory model from Week 4 Day 2.

Lecture Preview [0:00]

Today's agenda:

1. **Quiz 1 feedback** — Grading status, curve plans, test format reflection
2. **Assignment 2 Q&A** — Vehicle behavior clarifications
3. **Test-driven development** — Why write tests before implementation
4. **Technical debt** — The cost of "good enough" code
5. **Memory model continuation** — Complete the tracing exercise from Wednesday

Memory Model Continuation

The whiteboard demo from Wednesday ran out of time. Today we'll pick up where we left off and complete the full object instantiation trace.

Quiz 1 Feedback [0:39]

Grading Status:

- Tests are graded (raw scores posted)
- A few outstanding tests for emergency absences
- Question-by-question analysis still in progress
- Tests will be returned Wednesday with 10-20 minutes of in-class review

Test Length:

The test was too long. The goal is for the bulk of students to finish between 3/4 and 4/5 of the allotted time, with only 2-3 students remaining at the end. Instead, more than half the class was still working when time expired.

What Happened:

- Students got through the first part quickly (within 20 minutes, questions about the back half)
- Then hit a wall on the debugging/code-reading questions
- Many students left large portions blank

The Debugging Question Style:

The debugging-style questions were new. In hindsight, providing a practice question on Discord a few days before would have helped. (This was done for the TCSS 380 test with better results.)

> **Practice Recommendation**
>
> Spend time **hand-tracing** what code is doing. Be the compiler, but more importantly, be the runtime environment:
>
> - "I get to this statement—what exactly is it doing?"
> - "What is the output of this statement?"
> - "What becomes the input for the next statement?"

Grades and Curve:

- No traditional curve, but class average will be adjusted
- Everyone gets 4 points for a flawed question
- A Checkstyle question was worth 6 points but graded out of 4 (slight built-in curve)
- Potential take-home opportunity for commonly missed questions

Future Tests:

- Similar debugging/code-reading questions will appear again (with advance notice of question style)
- May include unit test analysis: "Given this spec and these 4 tests, which pass/fail?"
- Clear directions for bug fixes: "Point to the one thing that needs fixing"

Assignment 2 Q&A [15:41]

Human and Crosswalk Behavior [16:18]

Student Question

Q: I saw the human get run over. When does that happen?

A: Humans wait at green crosswalk lights (when vehicles have the right of way), then cross when the light turns yellow. A taxi can legally proceed through a yellow light—so if a human is crossing on yellow, the taxi can hit them.

Human behavior at crosswalks:

1. Light is green (for vehicles) → Human waits
2. Light turns yellow → Human starts crossing immediately
3. Taxi approaches yellow light → Taxi proceeds (can run over the human)

The only scenarios where a human gets disabled:

- ATV drives over them (ATVs ignore terrain rules)
- Human is in a crosswalk on a **yellow** light and gets hit

No street-driving vehicle goes through a **red** crosswalk light, so humans are safe once the light turns red.

Test-Driven Development Philosophy [19:01]

Student Question

Q: I finished the truck tests after writing the implementation. Writing the implementation first made more sense to me.

A: That makes complete sense—your brain is trained to write implementation code. That's what you've done since day one of 142.

The Training Problem:

Your programming education has trained you to:

1. Read a problem
2. Start writing implementation immediately

This works for simple problems. But as problems become more complex:

- "Just start coding" leads to more rework and refactoring
- The "throw spaghetti at the wall" approach becomes increasingly costly

The TDD Mindset Shift:

The goal isn't to completely reverse your thinking, but to introduce a new approach:

- **Stop and think** about the specification before implementation
- **Write tests** that verify documented behavior
- **Design first**, then implement

Student Question

Q: How are you supposed to think about methods before you write them?

A: You already do this—you read the document to understand the requirements. The real question is: how *much* should you think before writing? As problems get more complex, you need to invest more time in design. Once you've seen more patterns and implementations, you'll naturally spend more time thinking before coding.

Technical Debt [24:52]

The Scenario:

You write a method. It works. But the implementation makes you feel "icky inside." The tests pass. The deadline is tonight or tomorrow. You submit it.

For a class assignment that nobody touches again, this is fine (though it teaches bad habits).

In Professional Software:

1. **Sprint 1:** You submit that "icky" method. It works. You move on.
2. **Sprint 2:** New deadline. You build code on top of that method.
3. **Sprint 3:** More code built on top.
4. **Sprint 4:** You need to add new functionality, but the original method's design blocks you.
5. **Now:** To add the feature, you must redesign that method from Sprint 1—and rebuild everything built on top of it.

This accumulated cost is **technical debt**. Every piece of software built on that original bad decision carries 10%, 20%, 30% overhead.

> ⚠ **The Point Class Example**
>
> A student mentioned that Java's `Point` class has tech debt built in—the implementation is stuck and causes bugs everywhere it's used.

Build to Learn (Carefully):

It's okay to "build to learn." But be prepared:

- Build something to understand the problem
- **Tear it down completely**
- Rebuild it correctly from scratch

Don't just patch your learning prototype—you'll carry its design flaws forward.

Memory Model Continuation [30:36]

This section continues the memory model exercise from Week 4 Day 2, tracing through `memoryModelEx7()` with classes `A` and `B`.

Review: Load-Time Setup

Before runtime execution begins, the JVM performs load-time setup:

1. String Pool — The compiler identifies all string literals. At load time, these become `String` objects in a special heap region:

Address	Value
#123A	"Charles"
#123B	"Steven"

String objects in the pool are stored in contiguous memory addresses for efficient lookup.

2. Metaspace — `static final` constants are stored for fast access:

Class	Type	Name	Value
A	int	MAX_ID	1000

3. Class Objects — Every class used in the application has exactly one object representing it on the heap:

- `String.class`
- `Main.class` (or whatever the main class is called)
- `System.class`
- `Object.class`
- `A.class` — contains `idGenerator` (static, non-final)
- `B.class`

> ⚠ **Static Non-Final vs Static Final**
>
> - `static final` fields → Metaspace (constants)
> - `static` (non-final) fields → Live in the class object
>
> The `idGenerator` field is `static` but not `final`, so it lives in the `A.class` object, not in Metaspace.

Stack Frame Setup [32:24]

When `memoryModelEx7()` is called, a stack frame is created with space for local variables:

Type	Name	Value
A	a	(uninitialized)
String	sString	(uninitialized)

Stack frame sizing:

- The JVM knows exactly what local variables are needed before executing the method
- Primitive `int` = 4 bytes (32 bits)
- Object references = 8 bytes (64 bits on modern systems)—we're storing memory addresses, not objects

> **References, Not Objects**
>
> The `String sString` local variable doesn't store the string—it stores a memory address pointing to a String object. The size needed is always the same regardless of string length.

Tracing `new A("Charles")` [47:30]

```
A a = new A("Charles");
```

Step 1: What gets passed to the constructor?

Not the string "Charles"—the **memory address** of the string literal in the pool: `#123A`.

Step 2: Object instantiation

Memory is allocated on the heap for a new `A` object (address: `#619A`):

Field	Value
myName	(to be assigned)
myId	(to be assigned)

Step 3: Constructor execution

```
public A(final String theName) {
    super();
    myName = theName;
    myId = idGenerator++;
}
```

1. `super()` → Object constructor runs (implicit)
2. `myName = theName` → `myName` gets `#123A` (not "Charles"—the memory address)
3. `myId = idGenerator++` → Post-increment: read 1 from `A.class` object, assign to `myId`, then increment `A.class.idGenerator` to 2

Step 4: Assignment

The variable `a` on the stack now holds `#619A`.

Tracing Method Calls [58:25]

`a.toString()`:

```
return String.format("id: %d %nname: %s", myId, myName);
```

`String.format()` creates a **new** String object on the heap:

Address	Value
#791C	"id: 1\nname: Charles"

`sString` is assigned `#791C`.

`a.getName()`:

Returns `myName`, which is `#123A`—a reference to the pooled string. No new object created.

`a.convertName()`:

```
return myName.toUpperCase();
```

`toUpperCase()` creates a **new** String object:

Address	Value
#8111	"CHARLES"

What gets sent to `println()`? The memory address `#8111`.

`System.out.println("\n**********\n")`:

What gets sent? The memory address of the string literal in the pool: `#123C` (or wherever that literal lives).

Garbage Collection Interlude [1:03:03]

Student Question

Q: Is this like garbage collection?

A: No. Popping stack frames off the call stack is different from garbage collection.

Stack frames: When a method returns, its stack frame is "popped"—but nothing is actually erased. The stack pointer just moves. The data stays until overwritten by a new stack frame.

Garbage collection:

- Runs periodically (expensive operation)
- Looks for **orphaned objects**—heap objects with no references pointing to them
- Multi-step process: remove unreferenced objects, then remove objects only referenced by removed objects

Your Programs Probably Haven't Run Long Enough

Most student programs don't run long enough for the garbage collector to execute. It's designed for long-running applications.

After `convertName()` returns:

- `#791C` ("id: 1\nname: Charles") — still referenced by `sString`
- `#8111` ("CHARLES") — **orphaned** (no references)

If the garbage collector runs now, `#8111` gets cleaned up.

Tracing `new B("Steven")` [1:05:37]

```
a = new B("Steven");
```

A new `B` object is created. What are its instance fields?

Field	Value	Notes
myName	(inherited from A)	**Private** in A—B has it but can't directly access it
myId	(inherited from A)	**Private** in A—same restriction
myValue	(B's own field)	Directly accessible

> **Private Inheritance**
>
> Subclasses **have** the private fields of their parent—they're allocated in the object's memory. But they can't **directly access** them. That's why they're shown in red on the whiteboard.

Constructor chain:

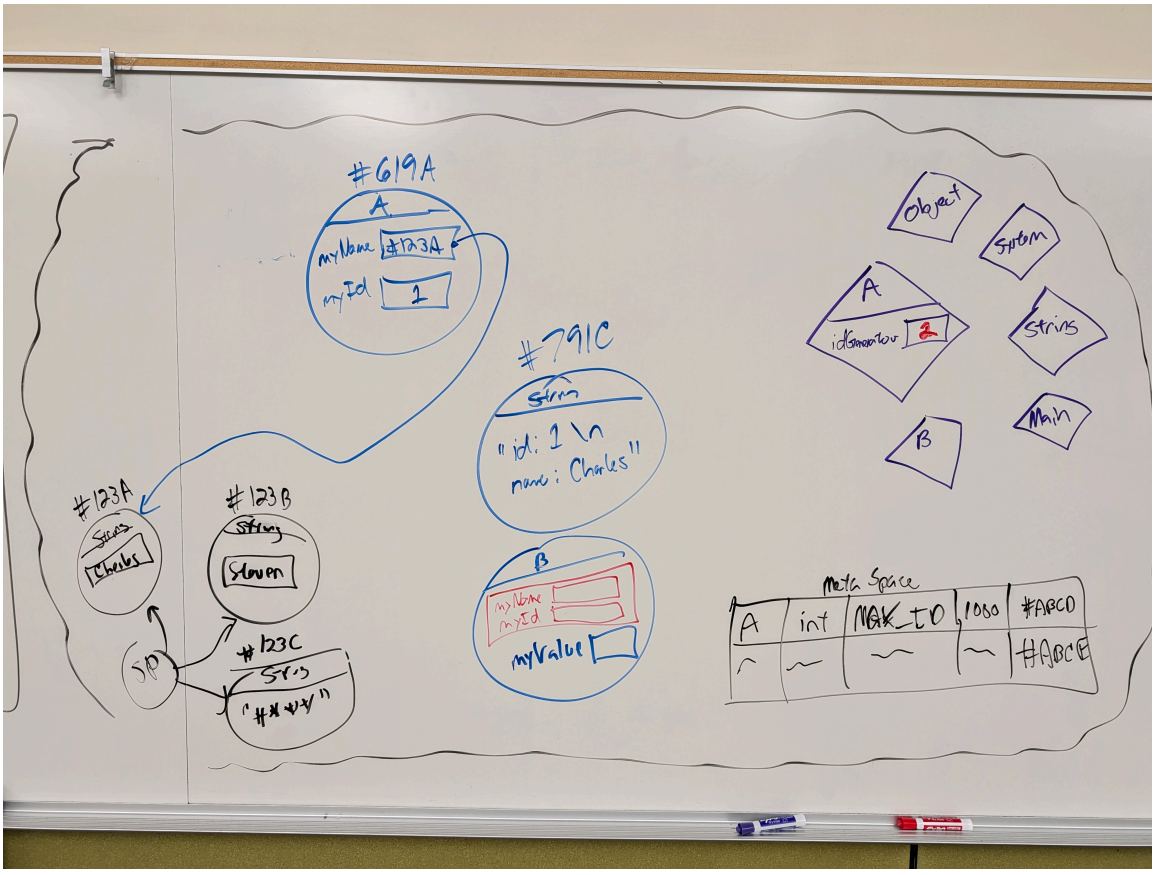
1. `new B("Steven")` passes `#123B` to B's constructor
2. B's constructor: `super(theName)` passes `#123B` to A's constructor
3. A's constructor: `myName = #123B`, `myId = 2` (idGenerator was 2, now 3)
4. Back to B's constructor: `myValue = theName.length()` → 6

> **To Be Continued**
>
> The lecture ran out of time here. Wednesday's lecture will complete the B object trace and show how polymorphism affects method calls on `a`.

Whiteboard Photos

Memory Model Heap Diagram

The complete diagram shows the heap (center/right) with object instances, class objects, and metaspace, plus the string pool (left).

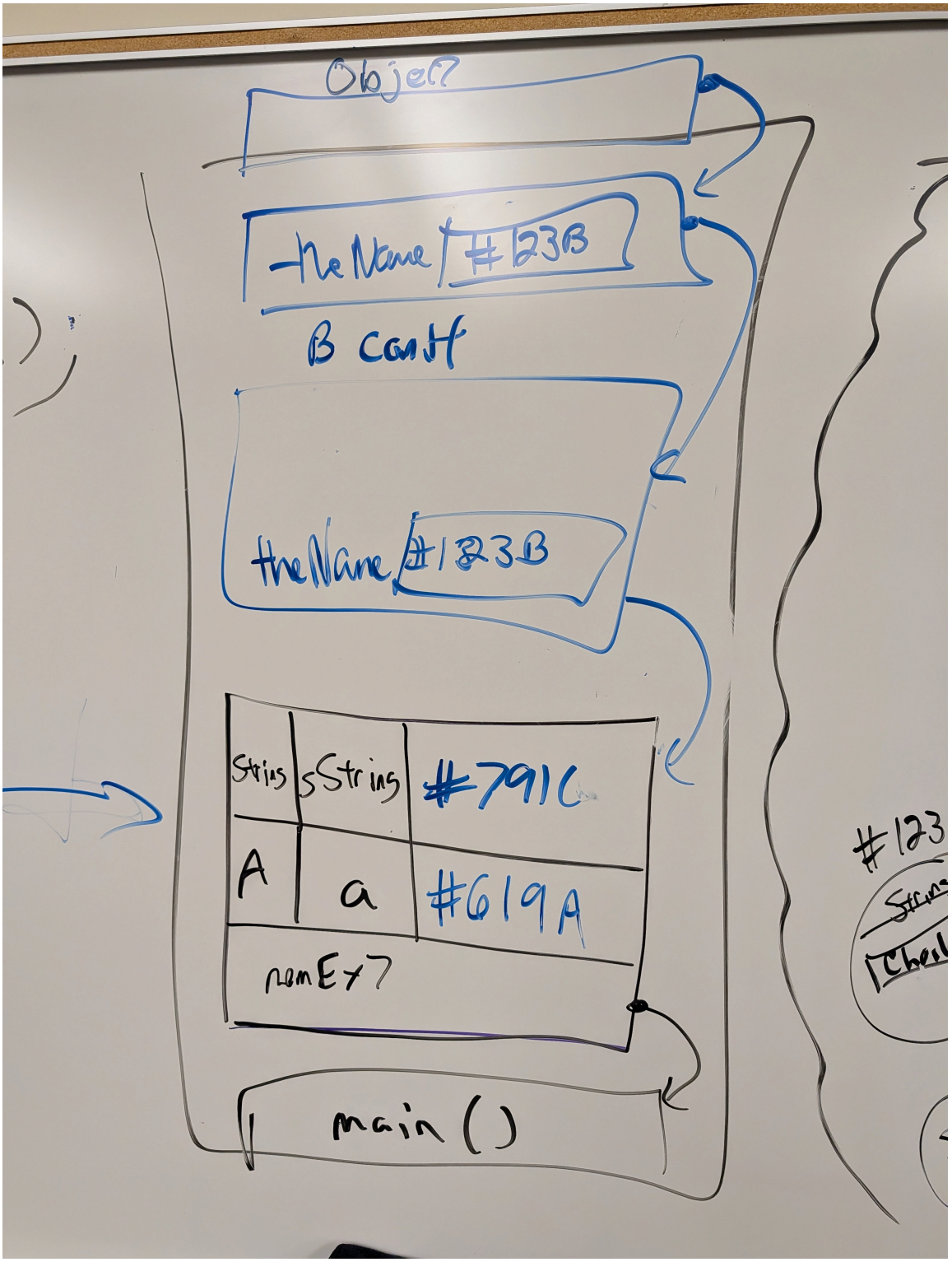


What's shown:

- **String Pool** (left): Map containing "Charles" at `#123A`, "Steven" at `#123B`, and the separator string at `#123C`
- **Heap** (center):
  - Object `A` at `#619A` with `myName` → `#123A` and `myId` = `1`
  - String object at `#791C` containing "id: 1\nname: Charles" (from `toString()`)
  - Object `B` with inherited fields `myName` / `myId` (in red—private) and `myValue`
- **Class Objects** (right): `Object.class`, `String.class`, `A.class` (with `idGenerator = 2`), `B.class`, `Main.class`
- **Metaspace** (bottom right): `A.MAX_ID = 1000`

Call Stack Diagram

This diagram shows the call stack during constructor chaining when creating the `B` object.



What's shown:

- **Stack** (bottom to top):
  - `main()` — entry point
  - `memEx7` — stack frame with local variables: `String sString = #791C`, `A a = #619A`
  - `A` constructor — `theName = #123B`

- `B Const` — B's constructor with `theName = #123B`
- `Object` — implicit `super()` call at top of chain

Lecture Demo Code

The following classes demonstrate the memory model concepts discussed in this lecture (same as Week 4 Day 2).

MemoryModelDemo.java

```
import edu.uw.tcss.mem.A;
import edu.uw.tcss.mem.B;

public class MemoryModelDemo {

    void main() {
        memoryModelEx7();
    }

    public void memoryModelEx7() {

        A a = new A("Charles");
        String sString = a.toString();
        System.out.println(sString);
        System.out.println(a.getName());
        System.out.println(a.convertName());
        System.out.println("\n***********\n");

        a = new B("Steven");
        sString = a.toString();
        System.out.println(sString);
        System.out.println(a.getName());
        System.out.println(a.convertName());
//        System.out.println(a.getValue());  // Won't compile - a is type A
    }
}
```

A.java

```
package edu.uw.tcss.mem;

public class A {

    /** Stored in Metaspace (static final constant). */
    private static final int MAX_ID = 1000;

    /** Stored in A.class object (static, shared by all instances). */
    private static int idGenerator = 1;
```

```
    /** Instance field - stored in each A object on heap. */
    private int myId;

    /** Instance field - reference to String (in pool or heap). */
    private String myName;

    public A(final String theName) {
        super();
        myName = theName;
        myId = idGenerator++;  // Post-increment: assign then increment
    }

    public String getName() {
        return myName;
    }

    public String convertName() {
        return myName.toUpperCase();
    }

    @Override
    public String toString() {
        return String.format("id: %d %nname: %s", myId, myName);
    }
}
```

B.java

```
package edu.uw.tcss.mem;

public class B extends A {

    private int myValue;

    public B(final String theName) {
        super(theName);  // Chains to A's constructor
        myValue = theName.length();
    }

    @Override
    public String convertName() {
        // Cannot access myName directly - it's private in A
        return getName().toLowerCase();
    }

    public int getValue() {
        return myValue;
    }
}
```
