# lectures # week-4

# Week 4: Memory Model Deep Dive (Wednesday, January 28, 2026)

**Lecture Recording**

Watch on Panopto

**Related Assignment**

This lecture covers concepts for Assignment A2 and builds on the memory model introduced in Week 1.

## Lecture Preview [0:00]

Today's agenda:

1. **Admin announcements** — Graduate program opportunities, Quiz 1 logistics
2. **Quiz 1 preparation** — Test format details, topic clarifications
3. **Assignment 2 Q&A** — Checkstyle warnings, testing specifications, vehicle behavior rules
4. **Memory model deep dive** — String pool, metaspace, class objects, and tracing object creation through memory

**Memory Model Continuation**

The whiteboard demo ran out of time and will continue Monday. The iPad whiteboard didn't connect, so photos of the physical whiteboard are included below.

---

## Admin Notes [0:57]

**4+1 Graduate Program:** CSE at UW Tacoma offers a 4+1 program where undergraduates can complete some master's requirements during their senior year, then finish the MS in one year instead of two. Details are on Discord—talk to advisors if interested.

**PhD Program:** UW Tacoma CSE has the only PhD program on the Tacoma campus. Several current PhD students came through TCSS 305/321. Undergraduate research positions are available—reach out to faculty if interested.

**Job Market Context:** The junior developer job market is challenging right now. Graduate degrees and research experience can differentiate you. Consider the long-term value of additional credentials.

---

## Quiz 1 — Final Details [15:05]

**Test Format:**

- Not fully assembled yet, but question bank is ready
- Some code writing (small snippets, not 20-line methods)
- Code reading and analysis
- Fill-in-the-blank (definitions and terminology)
- Matching (definitions and terminology)

**Grading Philosophy:**

- Target class average: B range (~85%)
- Curved if needed (zeros excluded from average calculation)
- Potential take-home makeup for commonly missed questions (half points back)

**Materials:**

- One handwritten note sheet allowed (not photocopied—you must write it yourself)
- Study groups can share notes for review, but everyone writes their own sheet

**Topic Clarification:** The `equals()`, `hashCode()`, and `toString()` methods from `java.lang.Object` (and the `java.util.Objects` helper class) are explicitly on the test. This was implicit in the topic list but is now explicit.

---

# Assignment 2 Q&A [22:47]

## Checkstyle: Use Interface Types for Variables

**Student Question**

Q: I'm getting a Checkstyle warning when I declare a `HashMap`. What's wrong?

A: Checkstyle wants you to declare the *variable* as the interface type (`Map`), not the concrete type (`HashMap`). You can still instantiate a `HashMap`.

```
// Checkstyle warning: use interface type
HashMap<Integer, String> myMap = new HashMap<>();

// Correct: declare variable as the interface type
Map<Integer, String> myMap = new HashMap<>();
```

**Why this matters:** Declaring the variable as the interface type (`Map`) means your code only uses methods defined in the `Map` interface. If you later need to swap `HashMap` for a different `Map` implementation (like `TreeMap` or a custom `FastMap`), you only change the instantiation—not every line that uses the variable.

**Rule of thumb:** Declare variables as high up the inheritance hierarchy as practical, without needing to cast constantly. Using `Object` for everything would work but requires casting everywhere. Using the appropriate interface (`Map`, `List`, `Set`) is the sweet spot.

**Polymorphic References**

`Map<Integer, String> myMap = new HashMap<>()` is a polymorphic reference. The variable type (`Map`) differs from the object type (`HashMap`). This pattern enables flexibility and is idiomatic Java.

## Law of Demeter [30:49]

**Student Question**

Q: I'm seeing Law of Demeter warnings in my code. What does that mean?

A: This rule warns against chaining method calls through multiple objects. It's disabled in our Checkstyle config—modern Java uses chaining heavily, so don't worry about it for this course.

This rule warns against chaining method calls like:

```
// Demeter violation: chaining through multiple objects
myMap.get(10).toLowerCase();

// Worse: deep chains are hard to read
myMap.get(10).getStudent().getGPA();
```

**The concern:** Long chains couple your code to the internal structure of multiple objects. If any intermediate type changes, your code breaks.

**Instructor's take:** This rule is disabled in the course Checkstyle configuration. Modern Java (especially Streams API) heavily uses method chaining. The rule made more sense in older Java styles. Be aware of the principle, but don't obsess over it for this course.

## Testing: Only Test What's Specified [36:41]

**Student Question**

Q: The Truck constructor takes `int x` and `int y`. Do I need to test for negative values or null?

A: No. Primitives can't be null (compiler error), and the spec doesn't define behavior for negative coordinates—so there's nothing to test. In graphics, negative coordinates are often valid anyway.

```
public Truck(int x, int y) { ... }
```

**Answer: No.** Primitives (`int`) cannot be `null`, so no `NullPointerException` test is needed. The specification doesn't say "throw exception for negative coordinates," so there's nothing to test. In graphics systems, negative coordinates are often valid (off-screen positioning).

**Test the Specification, Not the Implementation**

If the API documentation doesn't specify behavior for a particular input, don't write a test for it. You'd be testing your own implementation decisions, not the contract.

**Contrast with documented behavior:** The specification *does* say trucks have mass 100 and can only travel on streets/crosswalks/lights. Those are testable specifications. Tests should verify documented behavior.

### Vehicle Behavior: Taxi Crosswalk Rules [43:39]

**Student Question**

Q: The spec says the taxi "stops and waits for three clock cycles" at red crosswalk lights. Does that mean it can run the red after waiting?

A: Yes. After waiting three cycles at a red crosswalk light, the taxi proceeds through even if still red. If the light turns green before three cycles, it goes immediately.

**The spec says:** "For red crosswalk lights, the taxi stops and waits for three clock cycles."

**What this means:**

1. Taxi approaches red crosswalk, chooses to go straight (per its movement rules)
2. Checks light status → red → stays in place (count: 1)
3. Next cycle: chooses straight again → still red → stays (count: 2)
4. Next cycle: chooses straight → still red → stays (count: 3)
5. Next cycle: chooses straight → red → **already waited 3 times → proceeds through**

**Important:** If the light turns green before three cycles, the taxi goes immediately. The counter resets when the taxi reaches the *next* red crosswalk.

### Vehicle Behavior: "Reverse" Means Turn Around [47:16]

**Student Question**

Q: When the spec says vehicles "reverse" when blocked, does that mean driving backwards?

A: No—"reverse" means turn 180 degrees (about-face), not drive in reverse. A truck facing north with walls on three sides will turn to face south, then proceed forward on the next cycle.

---

## Memory Model Deep Dive [49:32]

This section traces through memory allocation using the demo classes `A` and `B`. The whiteboard diagram shows the complete memory layout.

## Memory Regions Overview

The Java memory model has several distinct regions:

| Region | What Lives There |
|---|---|
| **String Pool** | String literal objects (compile-time constants) |
| **Metaspace** | `static final` fields (constants) |
| **Heap (Class Objects)** | One object per class containing `static` (non-final) fields |
| **Heap (Instance Objects)** | Objects created with `new` |
| **Stack** | Method frames with local variables and parameters |

## String Pool [50:26]

When the Java compiler runs, it scans all code for string literals (like `"Charles"` and `"Steven"`) and builds a list. At runtime, these literals become `String` objects stored in a special area called the **string pool**.

```
String n1 = "abc";
String n2 = "abc";
n1 == n2  // true — same object in string pool

String n3 = scanner.nextLine();  // user types "abc"
n1 == n3  // false — n3 is a NEW String object, not from the pool
```

**Why `==` can be confusing with strings:**

- Two string *literals* with the same value → same object → `==` returns `true`
- A literal and a runtime-created string → different objects → `==` returns `false`

This seems non-deterministic if you don't understand the string pool. It's not random—it's about whether strings came from literals (pool) or runtime operations (new heap objects).

**Always Use .equals() for Strings**

Never use `==` to compare strings unless you specifically want to check if they're the same object in memory. Use `.equals()` for value comparison.

**String.intern():** The `intern()` method adds a string to the pool (or returns the existing pooled string if one exists). This is rarely needed—don't use it unless performance profiling demands it.

## Metaspace [59:00]

`static final` fields (class constants) are stored in **metaspace**, a special heap region for fast lookup.

```
public class A {
    private static final int MAX_ID = 1000;  // → Metaspace
    // ...
}
```

Metaspace entries include the class name, memory address, type, field name, and value:

| Class | Address | Type | Name | Value |
|---|---|---|---|---|
| A | #999A | int | MAX_ID | 1000 |

If a constant references an object (not a primitive), metaspace stores the reference, and the object lives in its own heap location.

**Student Question**

Q: Should all constants be static?

A: In Java, "constant" typically means `static final`. But you can have `final` instance fields (like `myName` in StoreItem) that are immutable per-object—those aren't really "constants" in the traditional sense, even though some programmers might call them that. Checkstyle enforces this distinction: it only allows `ALL_CAPS` naming for `static final` fields, not instance fields.

## Class Objects [1:00:06]

Every class loaded by the JVM has exactly one **class object** representing it. This is where `static` (non-final) fields live.

```
public class A {
    private static int idGenerator = 1;  // → Lives in A's class object
    // ...
}
```

The class object for `A` contains:

- A reference to itself: `A.class`
- Static method references like `getClass()`
- Static (non-final) fields like `myIdGenerator`

When you call `someObject.getClass()`, it returns a reference to this class object. When you write `A.class` in code, you're accessing this same object.

> **Where You've Seen .class**
>
> In JUnit: `assertThrows(IllegalArgumentException.class, ...)` — the `.class` syntax gives you a reference to the class object representing `IllegalArgumentException`.

## Stack and Heap: Tracing Object Creation [1:09:04]

Let's trace the first statement in `memoryModelEx7()`:

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

**Step 1: Stack frame setup**

The method gets a stack frame with local variables:

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

**Step 2: `new A("Charles")` executes**

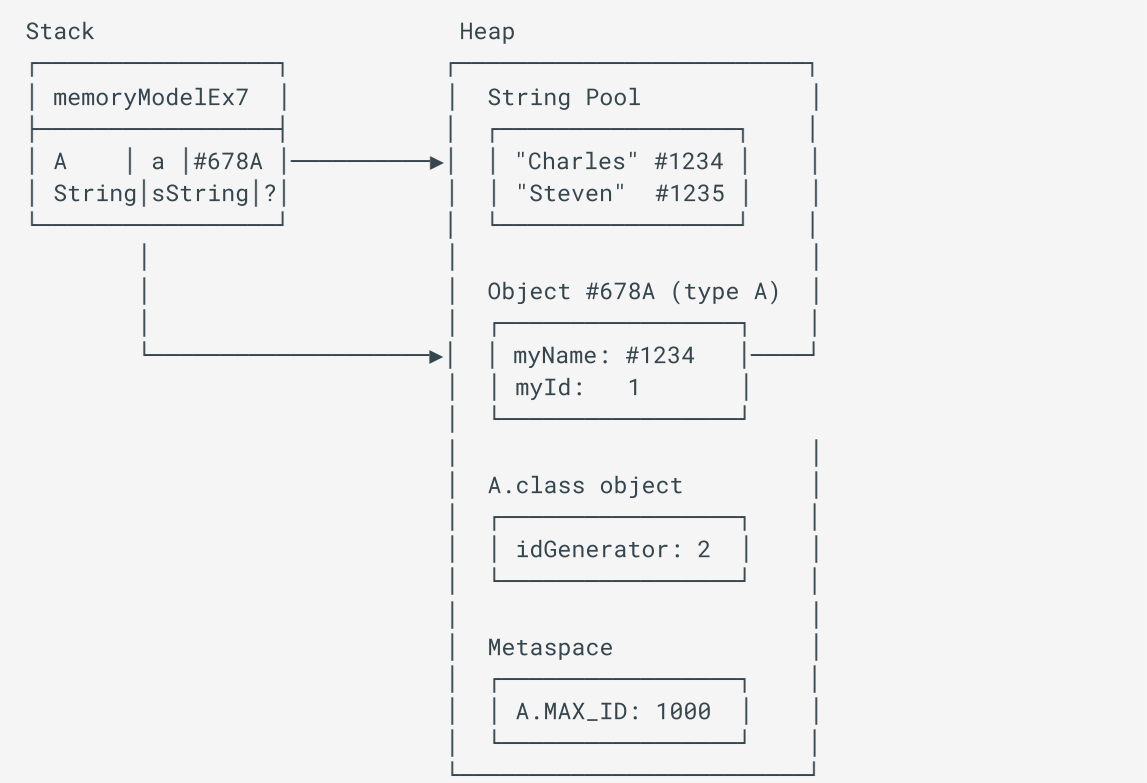
1. Memory allocated on heap for new `A` object (address: `#678A`)
2. `A` constructor runs:

3. `super()` calls `Object` constructor (implicit)
4. `myName = theName` → stores reference `#1234` (pointing to `"Charles"` in string pool)
5. `myId = idGenerator++` → reads `1` from class object, assigns to `myId`, increments class object's `idGenerator` to `2`

**Step 3: Reference assigned**

The variable `a` on the stack now holds `#678A`, pointing to the new object.

**Resulting memory state:**




> **To Be Continued**
>
> The lecture ran out of time at this point. Monday's lecture will continue the trace, showing what happens when `a = new B("Steven")` reassigns the variable to a `B` object (which extends `A`).

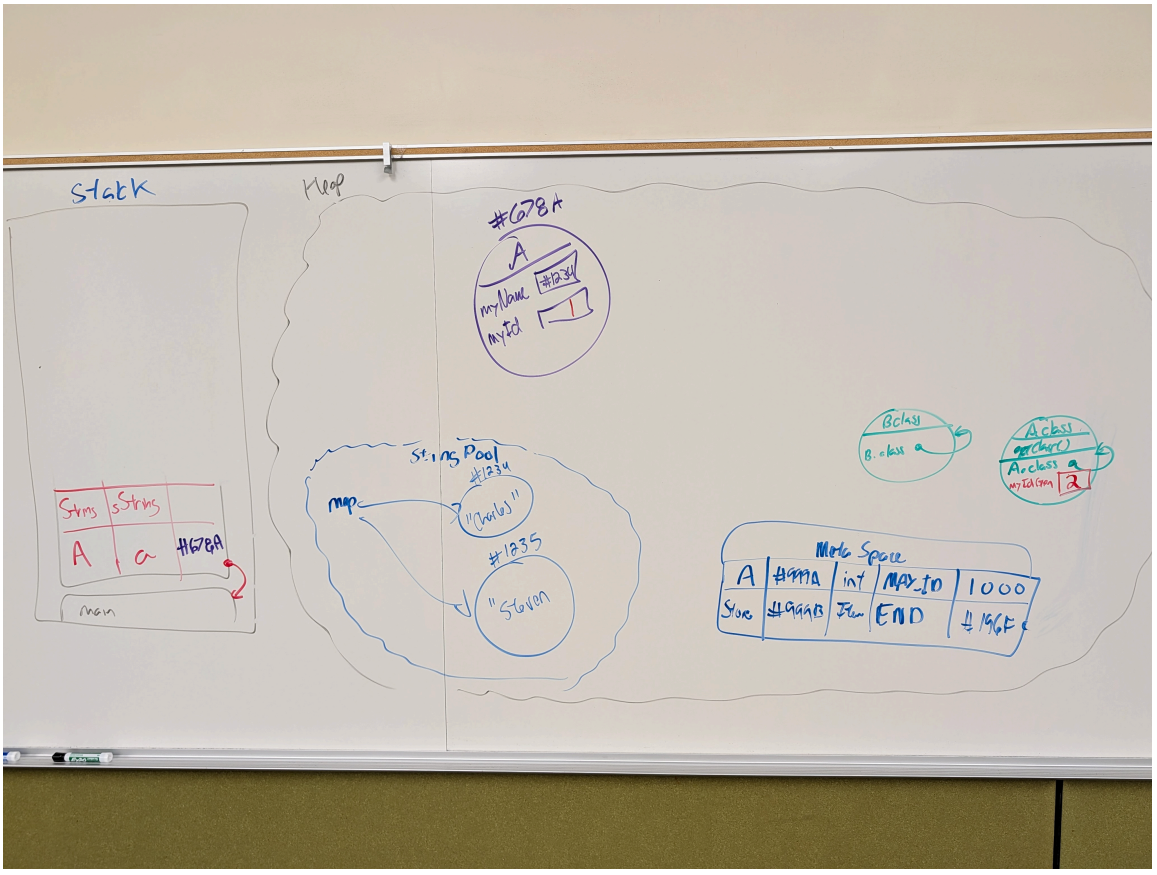
---

## Whiteboard Photos

The iPad whiteboard didn't connect, so here are photos of the physical whiteboard from class.

## Memory Model Diagram

The complete diagram shows the stack (left) with local variables, and the heap (right) containing the String Pool, object instances, class objects, and Metaspace.

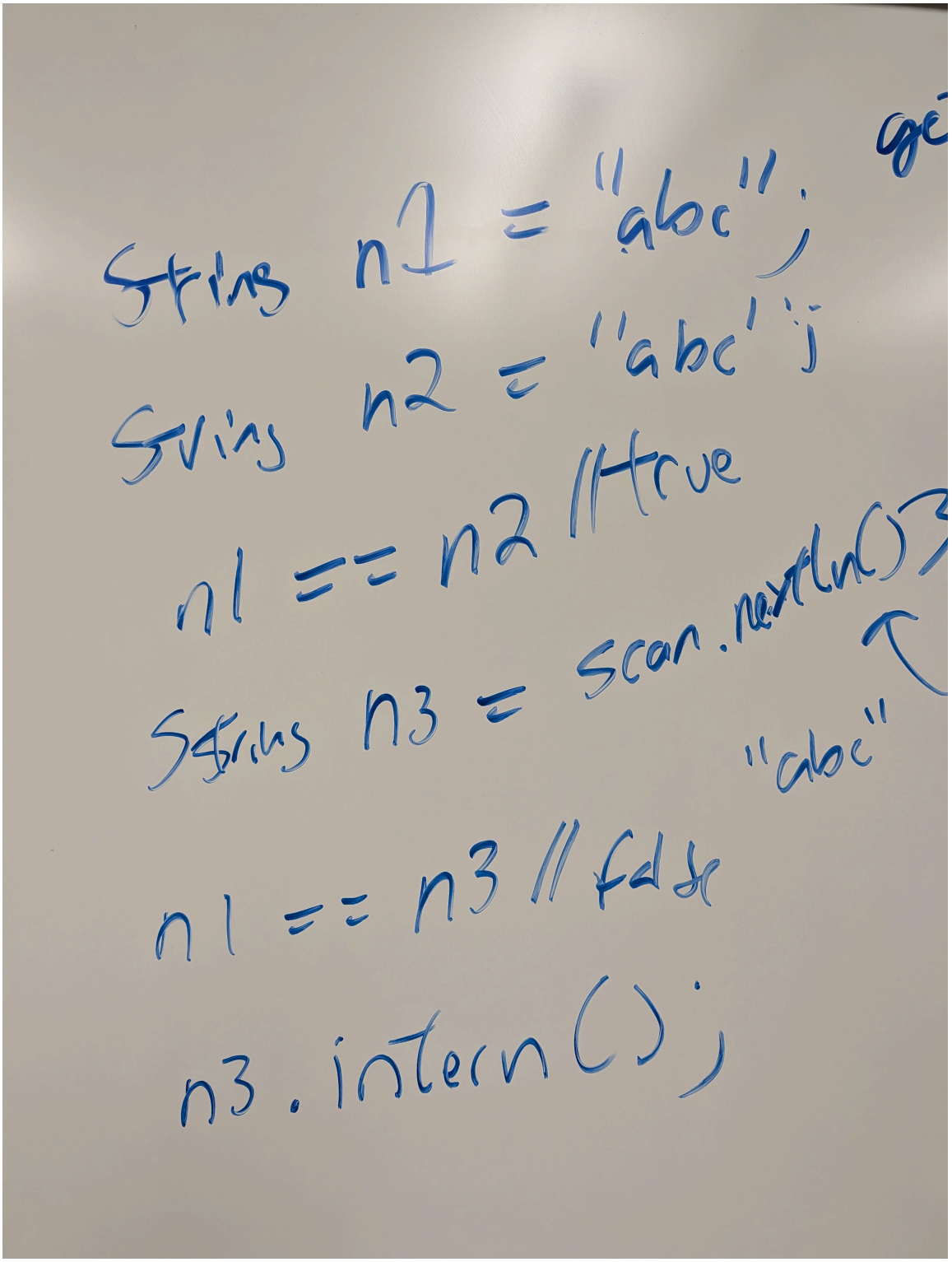



**What's shown:**

- **Stack** (left): Stack frame with local variables `a` and `sString`
- **Heap** (right):
- Object `A` at `#678A` with `myName` → `#1234` and `myId` = `1`
- String Pool with map containing `"Charles"` at `#1234` and `"Steven"` at `#1235`
- Metaspace with `A.MAX_ID = 1000`
- Class objects for `A.class` (with `myIdGen = 2`) and `B.class`

## String Comparison Example

This diagram shows why `==` behaves unexpectedly with strings—literals share objects in the pool, but runtime-created strings are separate heap objects.




**Code shown:**

```
String n1 = "abc";
String n2 = "abc";
n1 == n2  // true — same object in string pool

String n3 = scan.nextLine();  // user types "abc"
n1 == n3  // false — different objects
```

```
n3.intern();  // adds n3 to pool (or returns existing)
```

## Lecture Demo Code

The following classes demonstrate the memory model concepts discussed in this lecture.

**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;
    }
}
```

---

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