lectures # week-1

Week 1: Interfaces, Immutability, and Memory Model (Friday, January 10, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Assignment A1a and previews A1c.

Why Terminology Matters More Now [2:37]

You can probably write a class with instance fields, accessors, mutators, and a constructor. But can you *name* those things precisely?

Why this matters now more than 5 years ago: Gen AI will write the code for you, but you need to tell it what to create. If you say "create a class" and something's wrong, you need the vocabulary to explain what to fix.

- **Instance field** - a variable belonging to each object instance
- **Accessor** - method that retrieves state (not always `getX`)
- **Mutator** - method that changes state (not always `setX`)
- **Constructor** - initializes an object's state

Key Insight

"Be a designer. Be an architect. Be the computer scientist, not the programmer. The role of just coding is sliding away—you need to understand the concepts and speak the language precisely."

Interfaces as Contracts [13:02]

An interface defines the **public API** that implementing classes must provide.

What Can Be in an Interface?

Feature	Since	Notes
Abstract methods	Java 1.0	Public by default, no implementation
Constants	Java 1.0	`static final` fields
Static methods	Java 8	Can have implementation
Default methods	Java 8	Can have implementation (covered later)

For this course, focus on **abstract methods**—the contract that implementing classes must fulfill.

Two Parts of the Contract

1. **Syntactic contract** - The compiler enforces this. Does your class have all the required methods with correct signatures?
2. **Semantic contract** - The documentation defines this. Do your implementations actually *do* what the interface says they should?

```
// Syntactically correct but semantically WRONG
public String getName() {
    return null;  // Interface says "returns the name" - null is not a name!
}
```

Why Use Interfaces?

1. **Pure API definition** - Separates what from how
2. **Team coordination** - Team A implements, Team B uses—interface sits between them
3. **Guaranteed methods** - Compiler enforces the syntactic contract
4. **Polymorphism** - Treat different implementations uniformly (A1c preview: `StoreItem` and `StoreBulkItem` are both `Item`)

Key terminology: - **Client code** - Any code that uses another class's API - **Public API** - The public methods a class exposes

---

Implementing StoreItem [21:18]

To meet the semantic contract of the `Item` interface:

Step 1: Add Instance Fields

```
private final String myName;
private final BigDecimal myPrice;
```

Step 2: Accept Values in Constructor

```
public StoreItem(final String name, final BigDecimal price) {
    myName = name;
    myPrice = price;
}
```

Step 3: Return Actual Values

```
public String getName() {
    return myName;
}
```

Immutability: Make Variables Final First [27:30]

Default stance: Make all variables `final` at first. Only remove `final` when you have a genuine need to mutate—and then stop and ask *why* you need to change it.

Why? Over 50% of bugs in early code come from incorrect state. State becomes incorrect because we change variables.

Immutable Classes

If all instance fields are `final` and there are no mutators, the class is **immutable**—objects cannot change after construction.

```
public class StoreItem implements Item {
    private final String myName;      // Can't change
    private final BigDecimal myPrice; // Can't change
    // No setters = immutable
}
```

Example: `String` is immutable. Methods like `toUpperCase()` don't change the string—they return a *new* string.

> **Strive for Immutability**
>
> Immutable objects are easier to reason about, safer to share between threads, and less prone to bugs. If you can make a class immutable, do it.

Encapsulation: The Tylenol Analogy [31:07]

When you buy Tylenol, you know what it *does* (reduces pain, lowers fever) but not the exact chemical implementation inside the capsule. If the manufacturer changes the formula, your usage doesn't change.

Encapsulation = hiding implementation details behind a public API.

Why Encapsulation Protects YOU (The Developer)

Your code will evolve. After a year, you might realize a method is inefficient and want to rewrite it.

- **If fields are private:** You can change the implementation freely
- **If fields are public:** Client code may depend on those fields directly—changing them breaks everything

The `java.awt.Point` Horror Story [35:13]

This class has been in the JDK since Java 1.0 (1995). Look at its fields:

```
public int x;  // PUBLIC and MUTABLE
public int y;  // PUBLIC and MUTABLE
```

Everything wrong with OOP in one class: - Public fields = no encapsulation - Mutable fields = state can change unexpectedly - `int` precision = can't change to `double` without breaking everything

31 years later, Oracle still can't fix it because too much client code depends on those public fields.

Cardinal Sin of OOP

Public mutable fields would fail you in TCSS 143. James Gosling got away with it in 1994, and now the entire Java ecosystem is stuck with it.

Lesson: When you need a `Point` in your code, write your own properly encapsulated, immutable version.

Accessors vs Mutators (Not Getters/Setters) [40:46]

Prefer the terms **accessor** and **mutator** over "getter" and "setter":

- Not all accessors start with `get` (e.g., `calculateTotal()` accesses state)
- Not all mutators start with `set`

Pattern for immutable-style "mutation":

```
// Doesn't mutate - returns new object
public String toUpperCase() {
    return new String(/* uppercase version */);
}
```

When designing with AI, specify: "This class should be immutable. Mutating-like methods should return a new object."

Java Memory Model [43:06]

When a Java application runs on the JVM, it gets a region of memory divided into two areas:

The Call Stack

- Grows "up" from a meeting point
- Contains **stack frames** (one per method invocation)
- Each stack frame holds the method's **local variables** and **parameters**
- When a method completes, its stack frame is **popped** off

The Java Heap

- Grows "down" from the meeting point

- Contains all **objects** (created with `new`)
- Objects persist until garbage collected

What Goes Where?

Type	Location	What's Stored
Primitive variables (`int`, `boolean`, etc.)	Stack	The actual value
Reference variables (`Point`, `String`, etc.)	Stack	Memory address pointing to heap
Objects	Heap	Instance field values

The Swap Problem: Pass-by-Value [53:51]

Consider this code:

```
int x = 22;
int y = 99;
swap(x, y);
// x is still 22, y is still 99!
```

Why doesn't swap work?

Java is **pass-by-value**. When you call `swap(x, y)`:

1. Java looks up the *value* of `x` (22)
2. Java looks up the *value* of `y` (99)
3. Those *values* are copied into the method's parameters
4. The method swaps its local copies
5. Method ends, stack frame is popped
6. Original `x` and `y` are unchanged

You never pass the variable itself—only its value.

> **No Pointers in Java**
>
> In C, you can pass pointers to swap variables. Java doesn't have this. You cannot swap two primitives via a method call.

Object References in Memory [1:05:11]

```
Point p1 = new Point(10, 20);
```

What happens:

1. `new Point` - Creates a Point object in the **heap** at some memory address (e.g., `0xBEEF`)
2. **Constructor call** - Initializes `myX = 10`, `myY = 20` in that heap object
3. **Assignment** - Stores the memory address (`0xBEEF`) in the stack variable `p1`

Key insight: The stack variable doesn't contain the object—it contains a *reference* (memory address) to the object in the heap.

Instance Fields Per Object

Every object gets its own copy of instance fields:

```
Point p1 = new Point(10, 20);  // p1's object has myX=10, myY=20
Point p2 = new Point(22, 99);  // p2's object has myX=22, myY=99
```

A billion Point objects = a billion `myX` variables (one per object).

The Eight Primitive Types

Category	Types
Integer (5)	`byte`, `short`, `int`, `long`, `char`
Floating-point (2)	`float`, `double`
Boolean (1)	`boolean`

Note: `char` is numeric (it stores a Unicode code point). `String` is NOT a primitive—it's an object type.

Key Takeaways

1. **Know the terminology** - You need precise vocabulary to direct AI tools effectively
2. **Interfaces define contracts** - Syntactic (compiler-enforced) and semantic (documentation-defined)
3. **Make fields `final` by default** - Remove only when you truly need mutation
4. **Encapsulation protects you** - Public fields lock you into an implementation forever
5. **Java is pass-by-value** - Methods receive copies of values, not variables themselves
6. **Objects live on the heap** - Stack variables hold references (memory addresses) to heap objects

---

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