Lecture 8: Classes and Encapsulation

CS 2110

September 17, 2026

Today's Learning Outcomes

29. Given the description of a class, identify its state and behaviors and use these to sketch its fields and public method signatures.

30. Identify the invariant of a class and write code that maintains this invariant and/or asserts that it is satisfied.

31. Write an instance method given its specifications that accesses and/or modifies the values of its object's fields.

32. Draw memory diagrams that visualize the state of code involving instance method calls.

33. Determine the scope and lifetime of a variable.

34. Explain the object-oriented principle of *encapsulation* and its benefits. Evaluate the appropriateness of encapsulation in provided code.

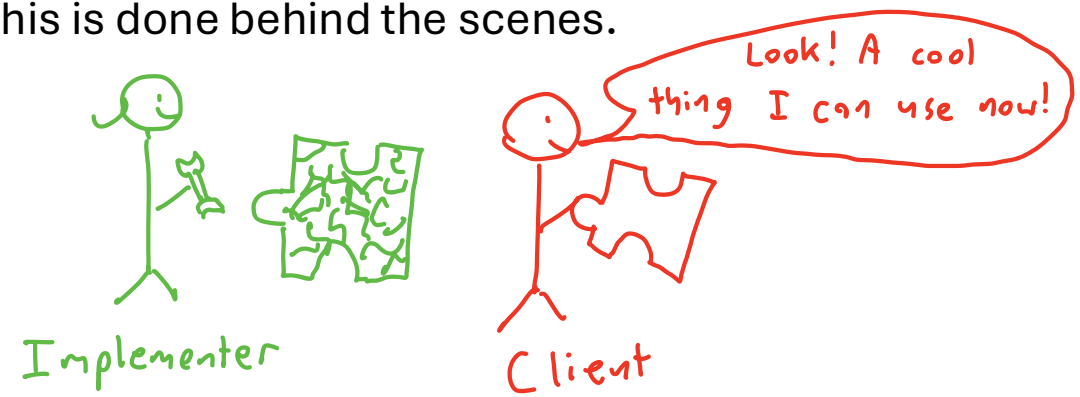
User-Defined Types

Recall: A **type** models a set of **states** (possible values) and **behaviors** (ways to use its data).

We'll often want to define a type that is custom-built for our application.

OOP and Abstraction

When we use objects in our programs, we can leverage their ability to model complex states and behaviors *without* having to worry about *how* this is done behind the scenes.

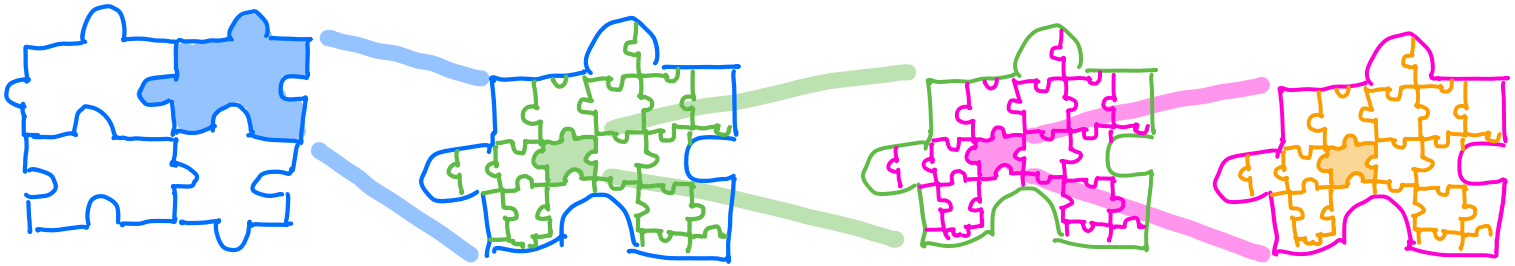


Writing Larger Software with OOP

Object-Oriented Programming with many layers of abstraction offers a principled way to write and maintain large software.

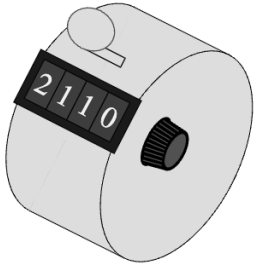
Modularization: Each class defines a clear set of responsibilities and promised behaviors (through specs)

Different developers work at different abstraction levels.



State Representation: Fields

Fields (or **instance variables**) are variables that model part of the state of an object.



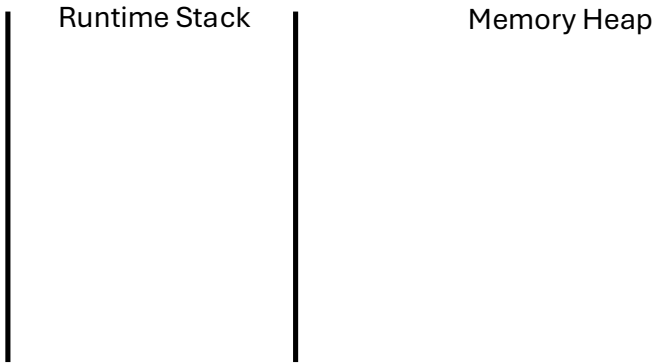
* Every field should have a Javadoc comment specifying how it should be interpreted, any requirements on its value, etc.

Construction

= the process of creating new instances of a class (i.e., objects)

```
public class Counter {
  int count;

  Counter() {
    this.count = 0;
  }
}
```

```
Counter c = new Counter();
```



Variable Scope and Lifetime

Two properties that describe the semantics of variables in our programs.

Scope

Lifetime

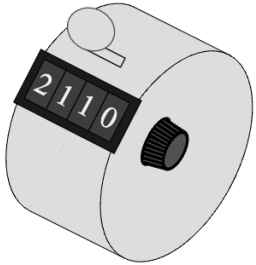
Variable Shadowing

We can sometimes have multiple variables with the same name in scope at the same time (as long as they're scoped to different levels).

```
public class Counter {
  int count;

  /** Updates `count` to the given value */
  void setCount(int count) {



  }
}
```

Behaviors: Instance Methods

Non-`static` methods declared within a class are its **instance methods**.

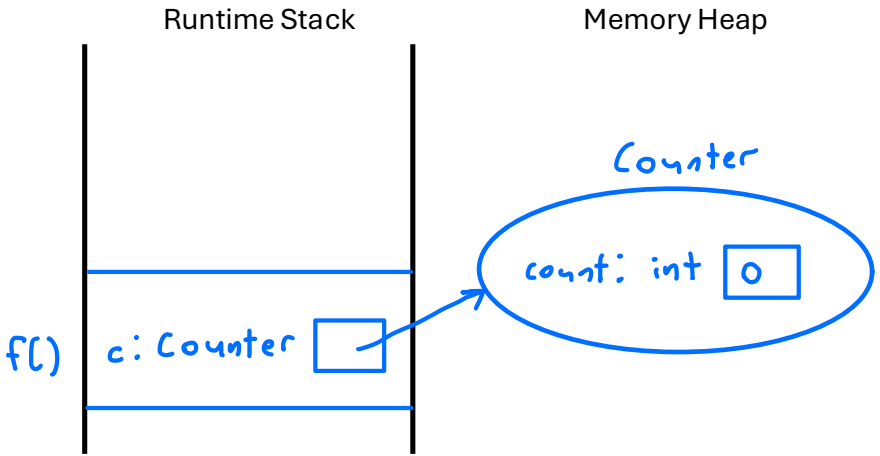


Diagramming Instance Methods

Same rules as before, but instance methods have an extra `this` entry that stores a reference to its target object.

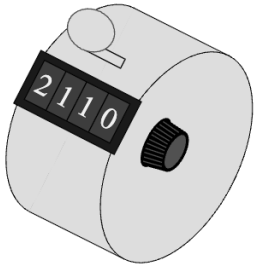
```
public class Counter {
  int count;

  void increment() {
    this.count++;
  }
}
```

```
static void f() {
  Counter c = new Counter();
  c.increment();
}
```



The Class Invariant

= a collection of properties of an object's fields that are defined in the field declaration Javadoc comments.



Devious Client Code

What will happen when we execute the following code?

```
1  Counter c = new Counter();
2  c.count = -13;
3  c.increment();
4
5  System.out.println(c.count());
```

Visibility Modifiers

We can add **visibility modifiers** in front of variable/method declarations to control from where (i.e., which class) they can be accessed.

```
public:


private:
```

Encapsulation

= a programming practice where we separate the concerns of the implementer and client of a piece of code.

The class invariant will be true every time that we enter/exit a `non-private` instance method of the class.

Another Example: Hangman

Guess letters to uncover secret word

- too many incorrect guesses "hangs man" and you lose

Coding Demo: Hangman Fields

New Ideas:

- `invariantSatisfied()` method as a **defensive programming** measure
- `final` keyword to prohibit field reassignment

Hangman Instance Methods

What behaviors should our Hangman class support?

Accessors:

Mutators:

Coding Demo: Hangman Behaviors

Defensive Programming:

- `assert` method pre-conditions
- `assert invariantSatisfied()` at end of mutator methods