Poll Everywhere

PollEv.com/javabear text `javabear` to 22333

Which of the following sorting algorithms has the best guarantee on its *best-case* asymptotic runtime complexity?

Insertion Sort $O(N)$ adaptivity		**(A)**
Merge Sort $O(N \log N)$		**(B)**
Quicksort $O(N \log N)$		**(C)**

URMC

UNDERREPRESENTED MINORITIES IN COMPUTING

SUPPORTING BLACK AND LATINX STUDENTS IN TECH THROUGH ACADEMIC SUPPORT, CAREER DEVELOPMENT, AND COMMUNITY BUILDING.

TA OFFICE HOURS

Come to do work sessions with fellow students in your classes, find new study buddies, and get help from URM TAs!

PRELIM REVIEW SESSIONS

Prepare for your upcoming prelims in review sessions led by URM TAs!

PROFESSIONAL DEVELOPMENT

We regularly host resume review events, internship apply-a-thons, and other events with our corporate sponsors!

MENTORSHIP

Get matched with a mentor who has been in your shoes and can help guide you through your journey at Cornell!

INSTAGRAM: @URMC_CORNELL
LINKTREE: HTTPS://LINKTR.EE/URMC
WEBSITE: URMC.CS.CORNELL.EDU/

JOIN OUR LISTSERV, GOOGLE CALENDAR, SLACK, AND CHECK OUT OUR WEBSITE!

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

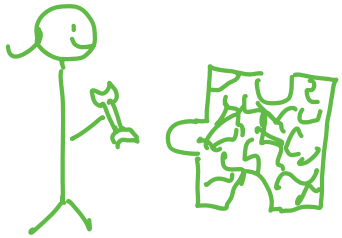
Recall: A **type** models a set of **states** (possible values) and **behaviors** (ways to use its data).

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

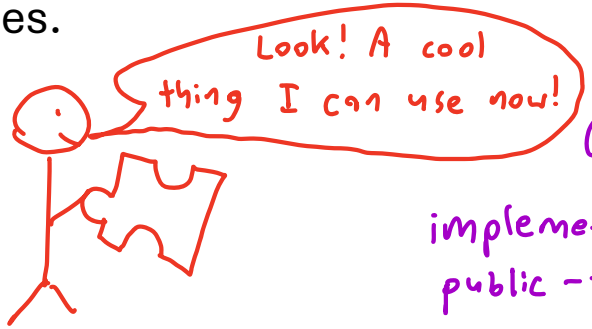
- Bundle together many related variables that track the state of a more complicated entity
  - User profile in application
  - Components (players, items, scenes, enemies) in game
  - Sophisticated data structures
  - ⋮
- Provide convenient methods to access and modify this state in a self-consistent way

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.



Implementer



Client

Abstraction = (mentally) separating implementation details from public-facing features

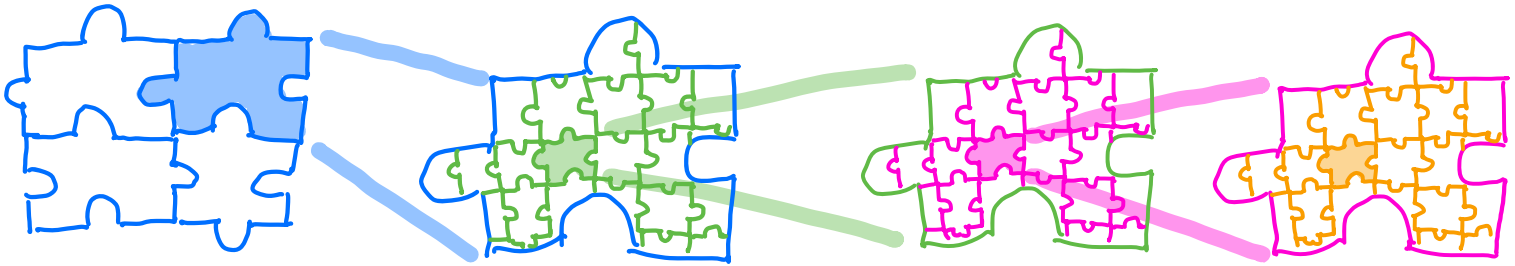
Object-Oriented Programming languages (like Java) organize code around defining new types and manipulating objects with these types.

Writing Larger Software with OOP

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

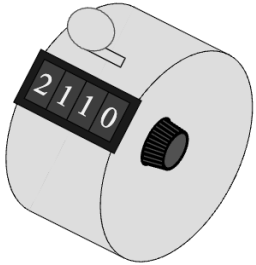
Modularization: Each class defines a clear set of responsibilities and promised behaviors (through specs)

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.



Declared within the class body but outside of any method

There is a separate copy of the fields created for each instance of the class. Fields live inside of objects

States

Possible configurations of an object

vs.

State Representations

Choice of fields to model those configurations

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

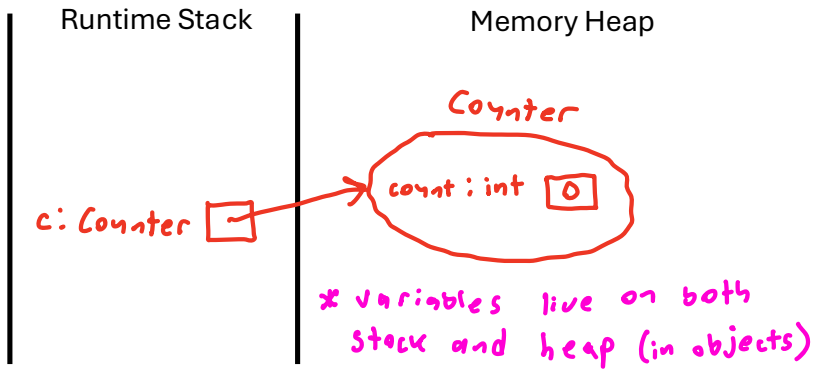
Construction

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

When you define a class with fields, add a constructor to initialize the values of these fields (otherwise they start with default values)

```
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

Describes where in the code variable can be used (accessed or assigned)

Method: Parameters + local variables
- from declaration through return

Block: Loop variables
- from declaration to end of {} block where declared

Class: Fields
- through any variable referencing object of that class (+ within class through "this")

Lifetime

Describes the time period between a variable's initialization and its deallocation

Stack frame entries have lifetime ending at method return

Fields have lifetime ended when we lose reference to object (and garbage collector erases it

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What can we say about the variable `i` immediately before we return from `frequencyOf()`?

```
1  static int frequencyOf(int[] a, int v) {
2    int count;
3    for (int i = 0; i < a.length; i++) {
4      if (a[i] == v) { count++; }
5    }
6    return count;
7  }
```

block scope ended

lifetime ends here

Within Lifetime? \ In Scope?	Yes	No
Yes	(A)	(B)
No	(C)	(D)

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).

Ex. local var/param + field with same names

Scope is resolved inside-out so local variable will shadow field.

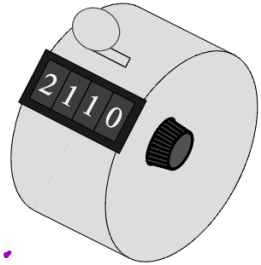
```
public class Counter {
  int count;

  /** Updates `count` to the given value */
  void setCount(int count) {
    this.count = count;
  }
}
```

this.count = count;
field param

Solution: access the field through this

Behaviors: Instance Methods

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



- Invoked on a target object of that type.

They can access/modify the fields of that object.

Accessor Methods

return info about objects state without modifying it

Mutator Methods

update fields

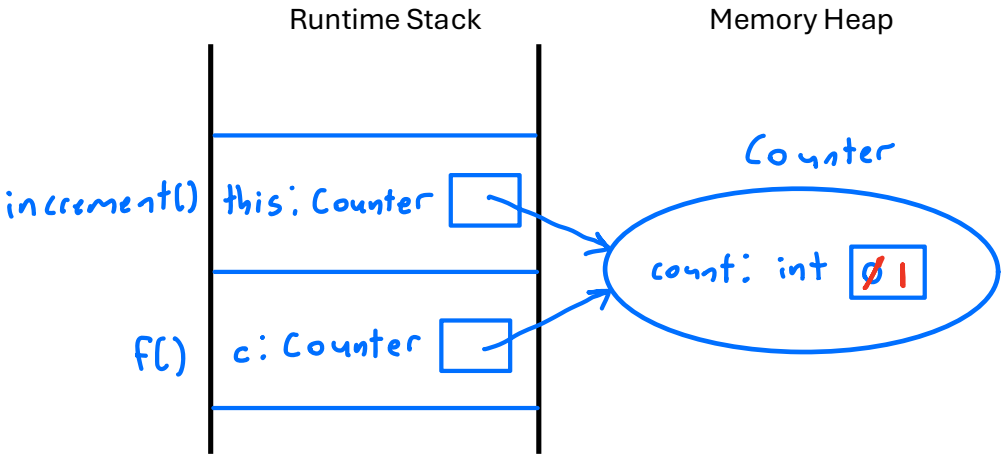
often run for side-effects

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();
}
```



Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What will be printed by the following code snippet?

```
1  Counter c1 = new Counter();
2  c1.increment();
3  Counter c2 = c1;
4  c1.increment();
5  c2.increment();
6  int n = c1.count();
7  c1.increment();
8
9  System.out.println(n);
```

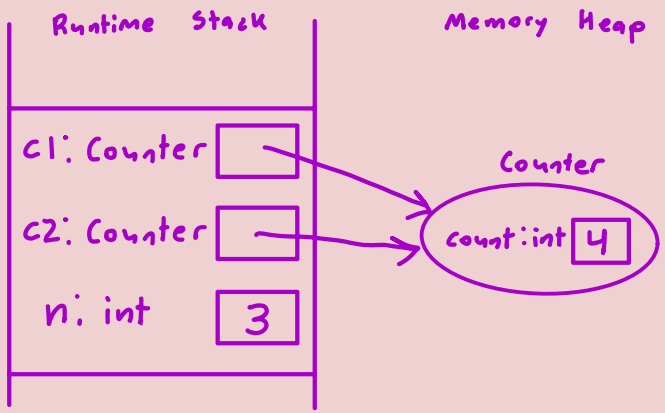
1	(A)
2	(B)
3	(C)
4	(D)

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

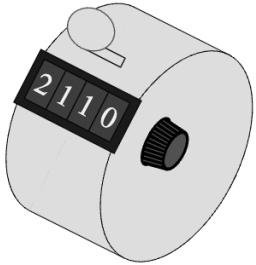
What will be printed by the following code snippet?

```
1  Counter c1 = new Counter();
2  c1.increment();
3  Counter c2 = c1;
4  c1.increment();
5  c2.increment();
6  int n = c1.count();
7  c1.increment();
8
9  System.out.println(n);
```



The Class Invariant

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



When the class invariant is true, the object is in a clean (i.e., self-consistent) state

Client should always see this

Implementer might temporarily need to "break" class invariant to carry out requested behaviors

Class invariant must be true on the way into/out of any instance method calls. *

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());
```

Using the current implementation, this code will run (in violation of class invariant).

c.count is a field with class scope, so it's accessible everywhere c is. We need some way, other than scope, to control variable access so we can protect our class invariants!

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`: visible from within any class (where they are in scope)
- good for many instance methods

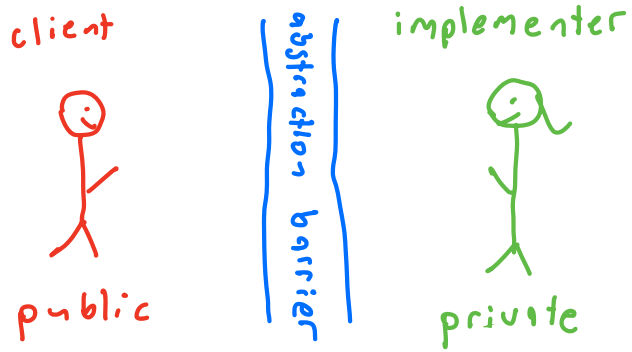
`private`: visible only from within class where field/method is declared
- good for fields / helper methods

default visibility: package private: Kinda weird... don't use anymore

Encapsulation

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

"strategic information hiding" - simplify view for client so they can't mess anything up



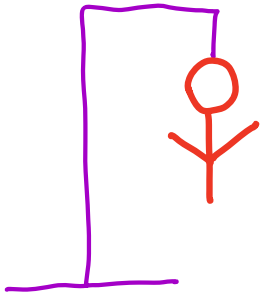
The class invariant will be true every time that we enter/exit a `non-private` instance method of the class.

Another Example: Hangman

Guess letters to uncover secret word

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

j a v a b e a r



s t l n

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:

- number of lives remaining
- previous guesses
- current "word state"
- whether game has ended
- secret word

Mutators:

- guess another letter

Coding Demo: Hangman Behaviors

Defensive Programming:

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