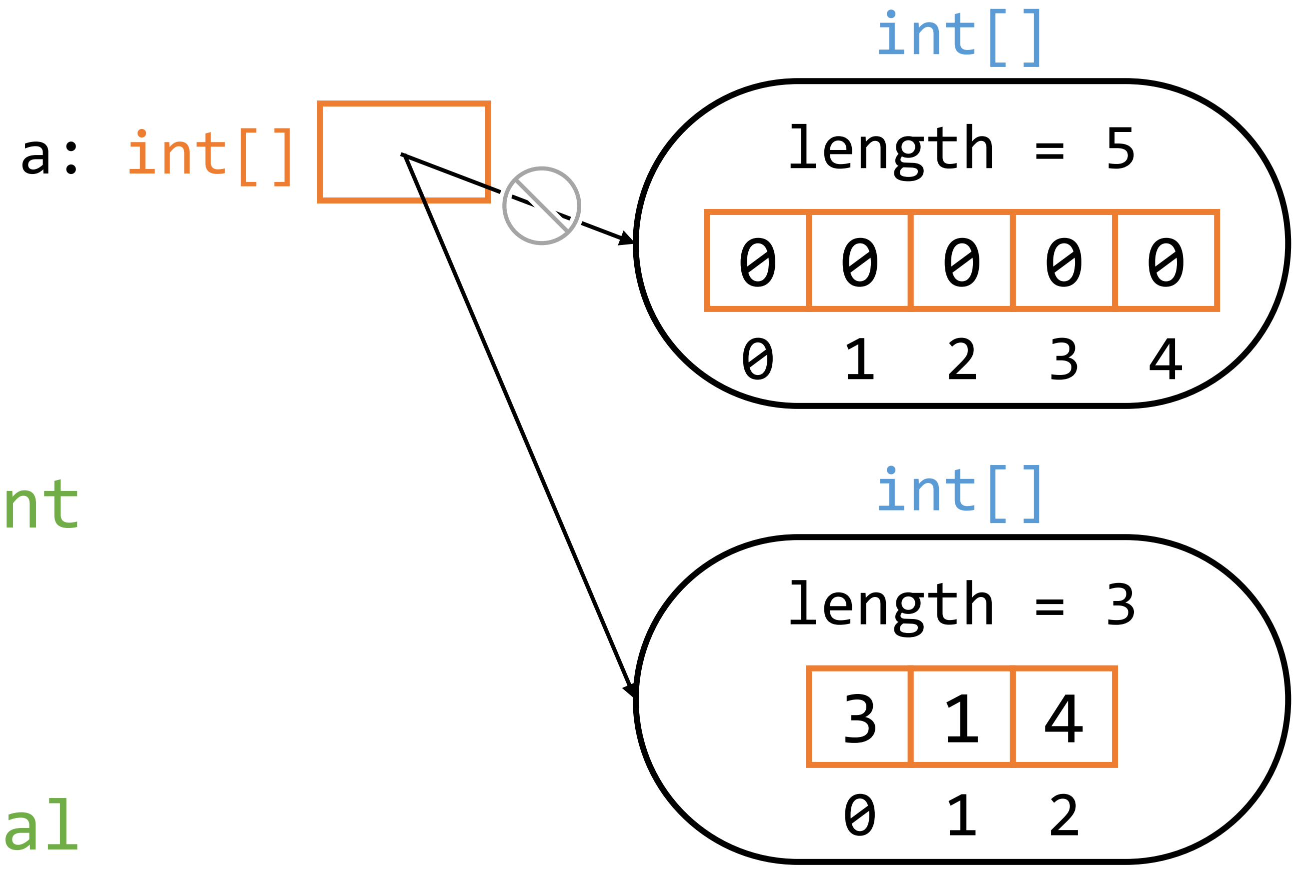
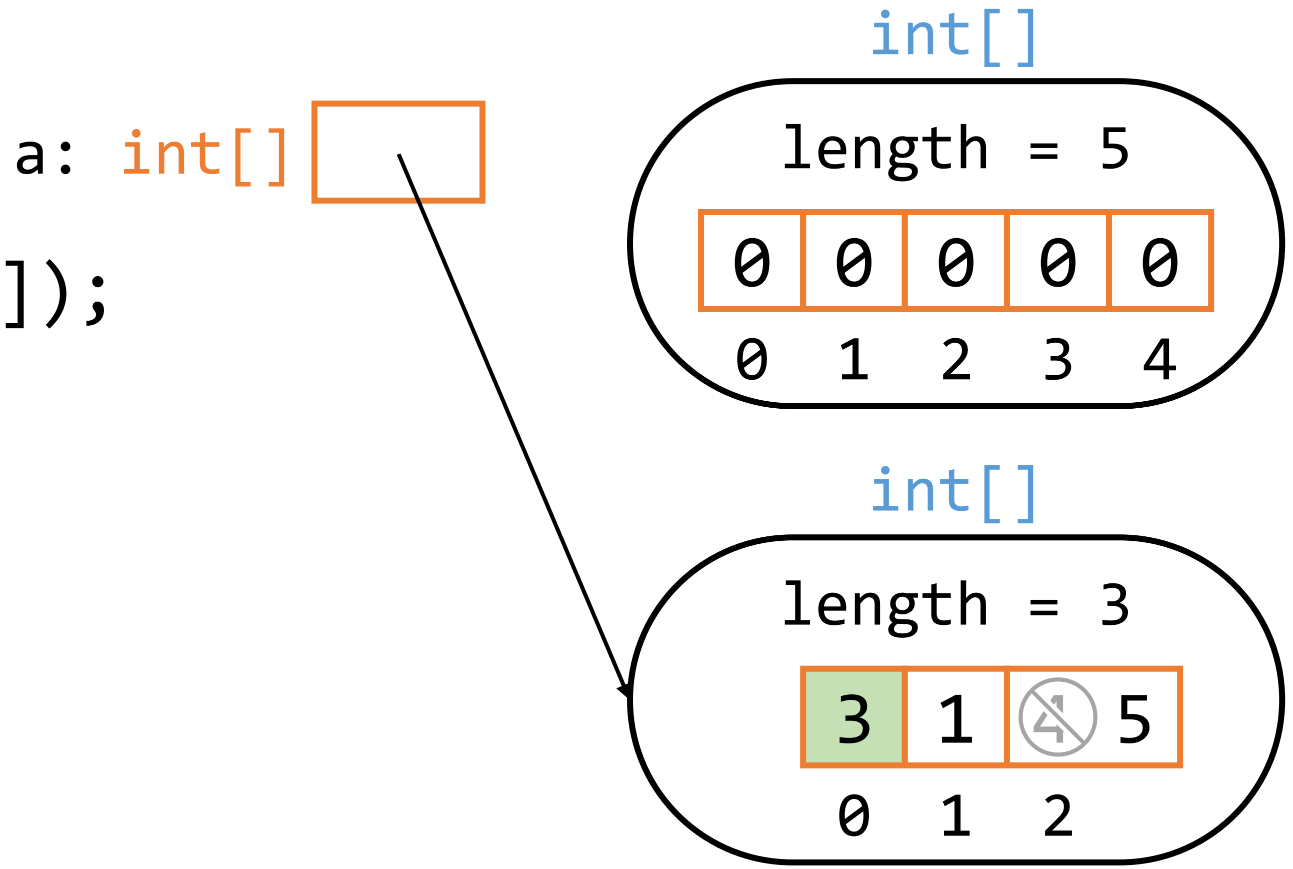
Arrays (not resizable!)

```
// Declaration
int[] a;

// Creation & assignment
a = new int[5];

// Assignment to literal
a = new int[] {3, 1, 4};
```



Array indexing

```
System.out.println(a[0]);
// Prints "3"

// Element assignment
a[2] = 5;
```



Warmup

- Draw an **object diagram** showing the program's memory after executing the following statements:

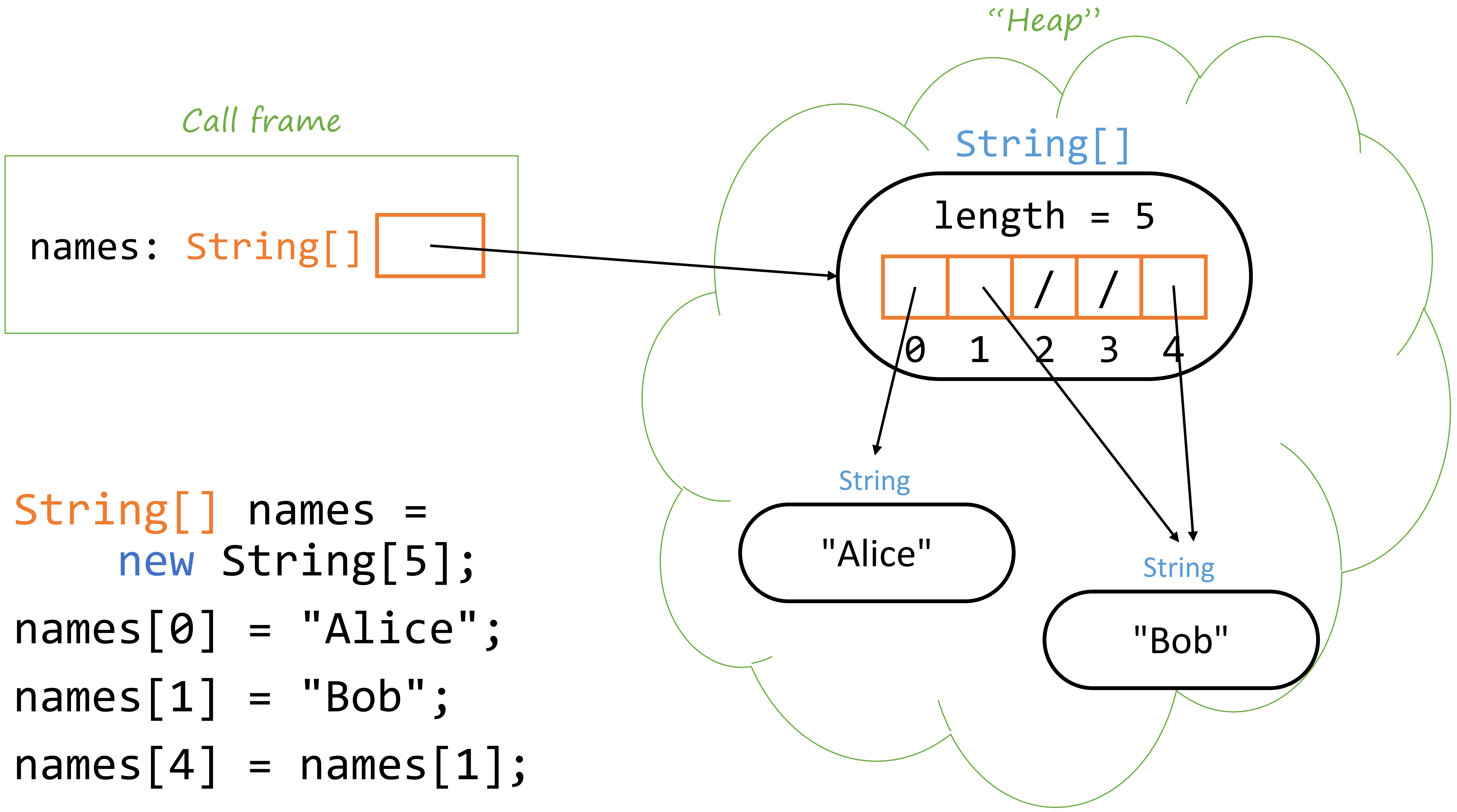
```
String[] names =
    new String[5];
names[0] = "Alice";
names[1] = "Bob";
names[4] = names[1];
```

Warmup

How many **objects** are in your diagram?

A. 1
B. 3
C. 4
D. 6
E. Other

```
String[] names =
    new String[5];
names[0] = "Alice";
names[1] = "Bob";
names[4] = names[1];
```



```
String[] names =
    new String[5];
names[0] = "Alice";
names[1] = "Bob";
names[4] = names[1];
```

CS 2110 Lecture 3

Class invariants, encapsulation

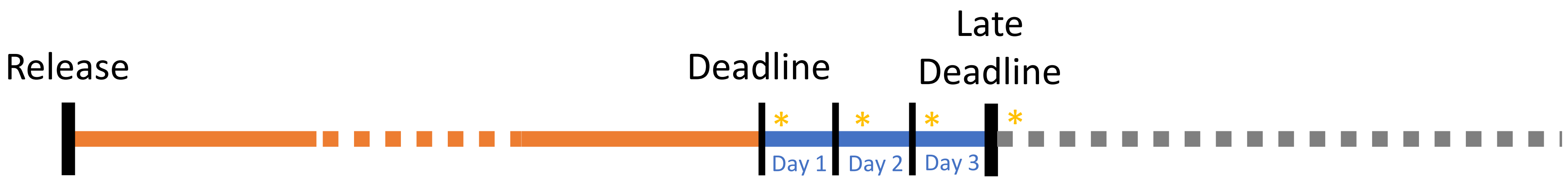
Reminders

A1 due tomorrow

Syllabus quiz

How slip days work

Slip days apply only to programming assignments A1–A6. See also "Student handbook" under References.



Submit to CMSX as often as you wish. No penalties. Make sure smoketests run!

Continue to submit as often as you wish. CMSX charges you one slip day as soon as the clock begins on a new day. No penalties applied yet.

No further submissions accepted in CMSX. No further slip days usable. During regrade period, may fix serious bug (10pt penalty) or complete an incomplete submission (25pt penalty).

End of semester: *timeliness* part of final grade based on total number of slip days used across all assignments. See syllabus.

*Short grace period (a few minutes), in case CMSX needs to restart.

Slip days are your own, self-granted extensions.
They are for illness, emergency, work in other courses, and so forth. Extensions beyond slip days are reserved for exceptional circumstances where slip days are insufficient (e.g., extended hospitalization), and requests for them should be accompanied by a letter from College Advising or SDS.

This lecture

- Invariants & constructors
- Access modifiers
- Defensive programming

Invariants and constructors

Object state

- Recall: A **type** is a set of values…
  - Which values are allowed?

- Is a Counter object with `counts=-1` valid?
- What about a dictionary that's not in alphabetical order?

Class invariants

- Invariant: a statement that should always be true
  - **Class invariant**: relationship between fields; truthfulness not affected by calling methods
    - Example: "\`counts\` is non-negative", aka `counts >= 0`
  - (**Loop invariant**: relationship between local variables, truthfulness not affected by loop iterations)
- Typically, invariants are expressed as comments; programmer is responsible for enforcing them
  - Can write code to *check* them – catches bugs
  - (Some languages can verify invariants; see Dafny)
- Should also specify how fields should be interpreted

Example: Counter

- State representation:
  `int counts;`
- Allowed states:
  [-2147483648, 2147483647]

- Which states *should* be allowed?
- Could implementations of behavior yield disallowed states?

Demo: Improving Counter

Types and invariants

- **Static typing** enforces common invariants automatically!
  - “`int counts` *is an integer*”
  - “`String name` *is a string of characters*”
- If an invariant concerning a field is captured in the field’s type, you *do not* need to document it separately
  - Comments redundant with code are a maintenance burden

Creating objects

- How to initialize fields to represent a state *specified by a user*?
- How to establish that the *class invariant is satisfied* from the start?

Constructor

- Syntax
  - Like a method, but no return value, and name matches name of class
  - Invoked with new-expression
  - Can delegate to other constructors by calling `this()`
- Job: truthify the class invariant
  - Initialize all field values
- Default constructor
  - No parameters
  - Initialize all fields to default values

```
class Counter {
    int counts;
    Counter() {
        counts = 0;
    }
    // ...
}
```

Constructor syntax example

```
class Point {
  final double x;
  final double y;

  Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
}
```

```
// Client code: invoke
// constructor in new-expr.
Point p = new Point(1, 2);
```

Mutability

Immutable types are not distinguished in Java; must rely on documentation

Variables

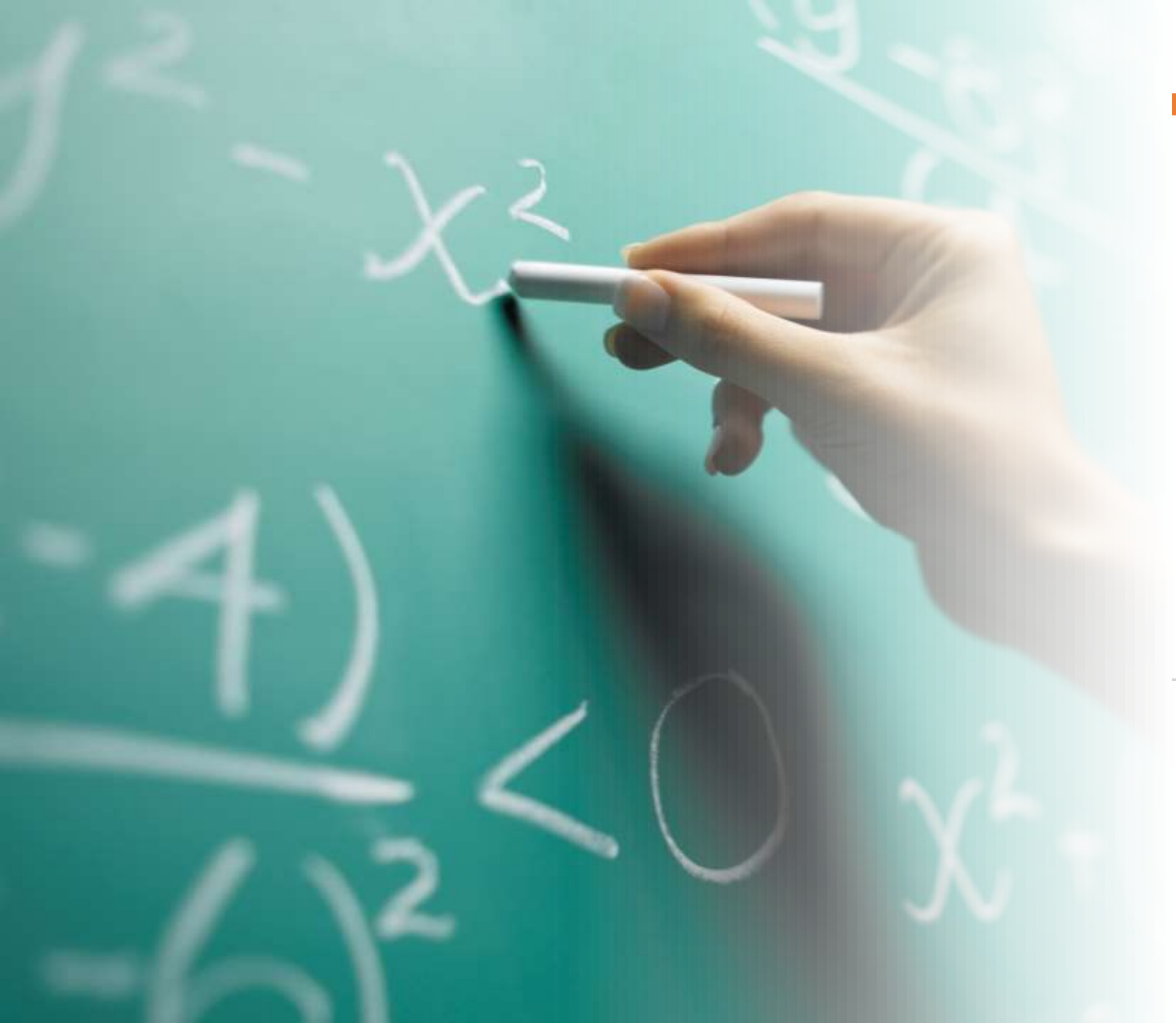
- By default, variables can be reassigned
  - `int x = 0;`
    `x = x + 1;`
- A `final` variable can only be assigned to once
  - `final int serialNum = sn;`

Types

- Values of **immutable** types cannot change their *state*
  - Mimic value semantics
  - Variables change value through reassignment
  - Example: `String`
- Values of **mutable** types can change their state
  - Side effect of methods (reassign or mutate fields)
  - Changes visible through all aliases
  - Example: `Counter`

Example: Fraction

- What fields could represent a fraction?
- Are any field values invalid?
- If representing points on number line, are any field values redundant?



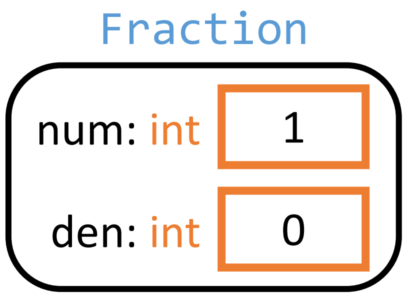
Demo: Developing a Fraction class

Encapsulation

Fractions continued

```
/** Represents a rational
  * number. */
class Fraction {
  /** The numerator of the quotient
   *  representation `num/den`. */
  int num;

  /** The denominator.  Must
   *  be positive, and the GCD
   *  with `num` must be 1. */
  int den;
  ...
}
```

- Is this Fraction object a bug?

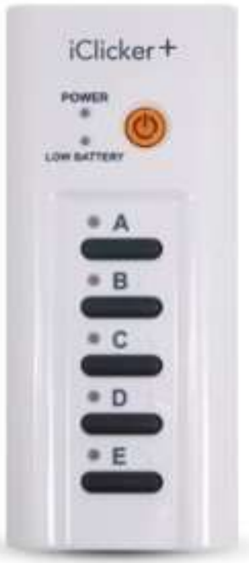


Demo

```
Fraction f =
    new Fraction(0, 1);


f.den = 0;


println(f.toDouble());
```

- Whose fault is this?

  A. The person who wrote Fraction.java
  B. The person who wrote `f.den = 0`



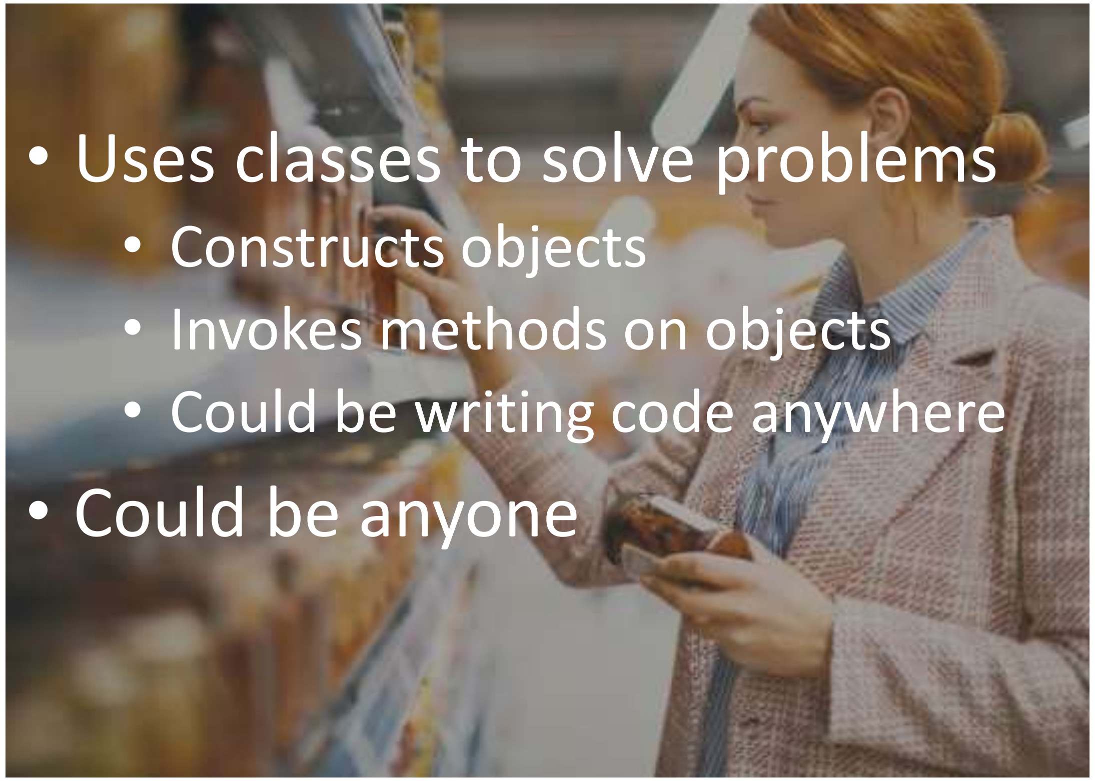
Fun fact

Opening IntelliJ involves interactions between objects from…

- 600 different classes?
- 6,000 different classes?
- 60,000 different classes!

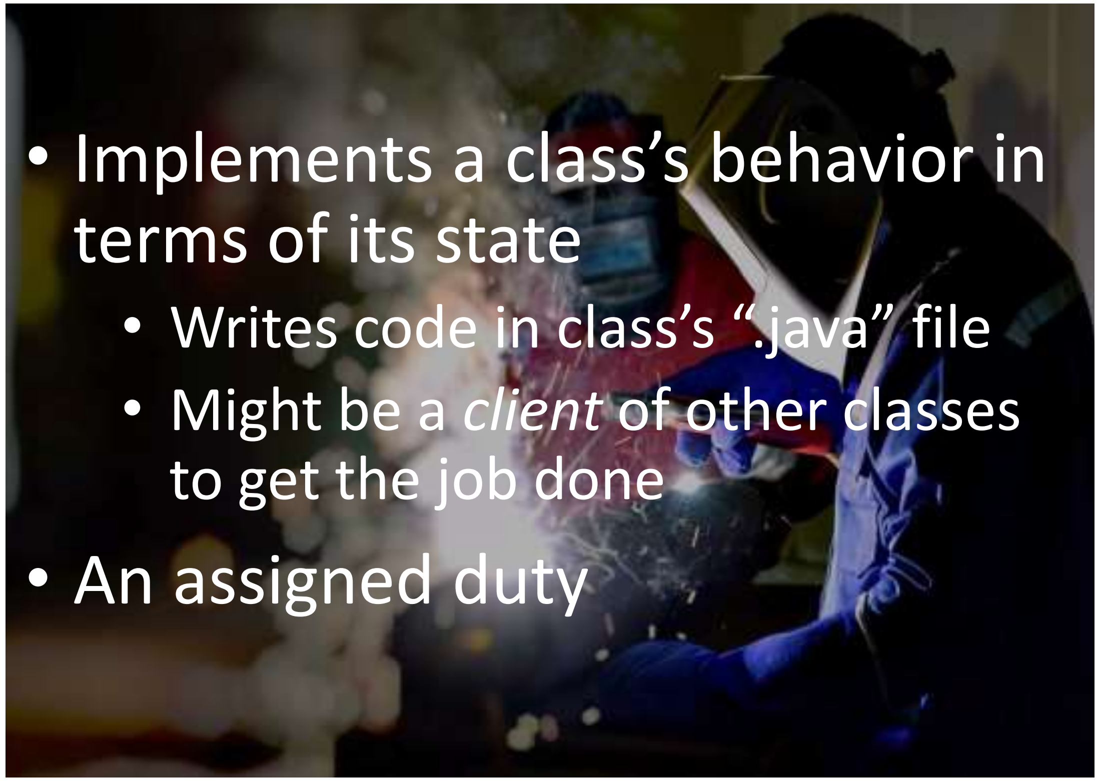
Dividing responsibilities

Client



- Uses classes to solve problems
  - Constructs objects
  - Invokes methods on objects
  - Could be writing code anywhere
- Could be anyone

Implementer



- Implements a class's behavior in terms of its state
  - Writes code in class's ".java" file
  - Might be a *client* of other classes to get the job done
- An assigned duty

Client vs. implementer

- Refers to a **role** with respect to a class in a particular context
    - We are all *clients* of the String class
    - None of us is the *implementer* of the String class

    - I am the *implementer* of my Fraction class when writing code in "Fraction.java"
    - I am the *client* of my Fraction class when writing code in "FractionDemo.java" (or anywhere else)
- You will serve both roles, sometimes for the same type
    - Practice "splitting your brain" to adopt the appropriate role

Responsibilities

- Implementer must maintain class invariant, provide correct behavior

- Client should be able to use class *for any purpose* and never get incorrect behavior

How can implementer prevent clients from breaking things?

Encapsulation

- Programming languages can help us protect a class's state
  - What if state were invisible to users? What if they could only invoke (a subset of) objects' behaviors?
  - Theme: giving up flexibility to achieve reliability
- **Access modifiers**
  - `public`: Anyone can access fields / invoke methods
  - `private`: Only the class implementation can access fields / invoke methods

Encapsulated Counter

```
public class Counter {
    /** Class invariant: `counts` is in [0,9999]. */
    private int counts;

    public Counter() { counts = 0; }
    public int getCount() { return counts; }
    public void reset() { counts = 0; }
    public void increment() {
        if (counts == 9999) { counts = 0; }
        else { counts += 1; }
    }
}
```

Accessibility recommendations

- If class is public, fields should always be private
- Public methods provide meaningful behavior to clients
  - Maintenance burden: cannot change behavior without breaking clients
- “Helper” methods should be private

OOP language features

Encapsulation	Polymorphism	Inheritance
• Lecture 3	• Lecture 5	• Lecture 6

Encapsulation

- Procedural programming: data is “in the open”, accessible by code from anywhere in the program (written by different people)
  - Must trust all code to preserve invariants associated with all data
  - To change data layout, must update code everywhere

- OOP bundles data with associated behavior, prevents direct access
  - Only class implementer must be trusted to preserve invariants
  - Class implementer can change data representation without changing any code outside of class
  - Decouples code using an “abstraction barrier”

Anti-pattern: getters/setters

- Many Java classes mark their fields as private, then declare public “getter” and “setter” methods to provide read and write access
  - Wrong mentality: focus is on state rather than behavior
  - Think in terms of “observers”, not “getters”; no need for “`get`” prefix

- Encapsulation is about **abstraction**, not just protection
  - Best practice: choose behaviors before state representation