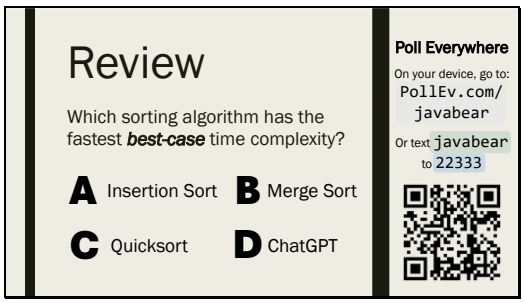
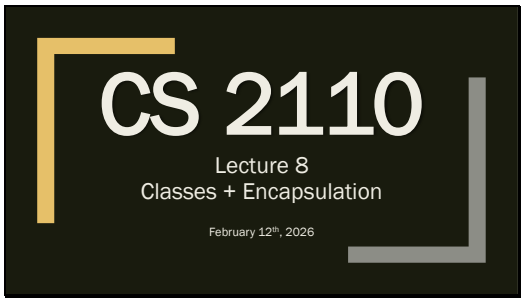
Slide 1

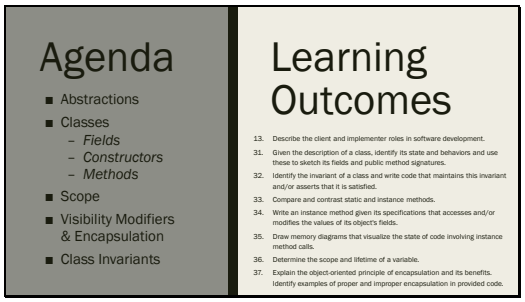


Correct Answer: A
Insertion sort is <u>adaptive</u>, with a better complexity in its best case of $O(n)$.
Merge sort is $O(n \log n)$ in all cases (advantageous if you're trying to avoid the worst case), and quicksort's best case is also $O(n \log n)$ but has a worse worst-case $O(n^2)$ compared to merge sort (but it has better constants).
ChatGPT is good for hallucinating the wrong sort order.

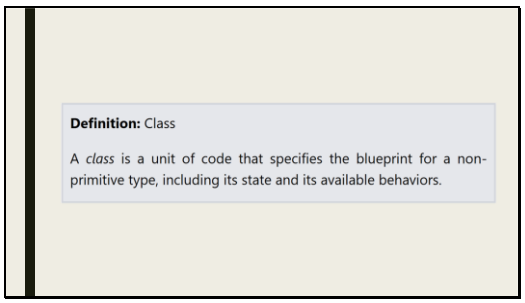
Slide 2



Slide 3

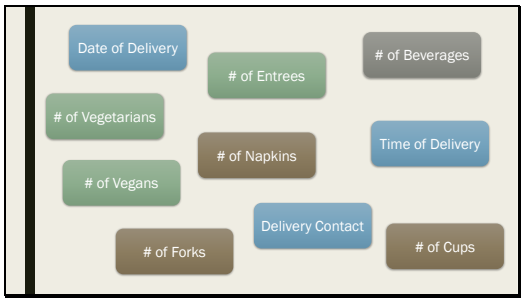


Slide 4



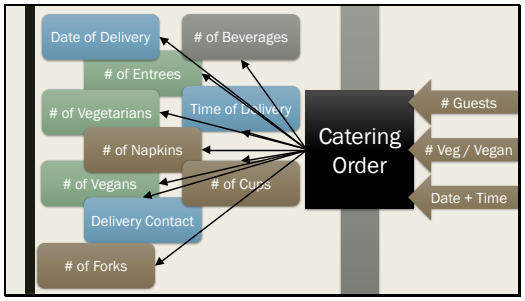
Back in lecture 2, we saw the definition of a class, which talks about a *type* that includes *state* and *behavior*. But if you were anything like me, this definition didn't really mean anything at the time. So, if you'll bear with me, I'm going to start by telling a (admittedly not very interesting) story.

Slide 5



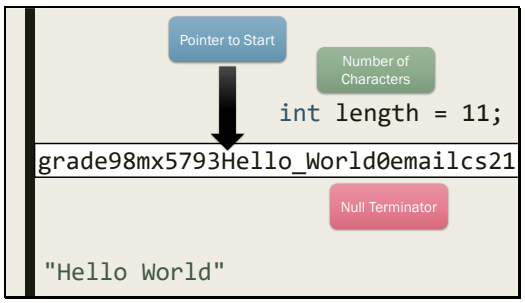
The first year I was a TA, I got roped into being the person in charge of ordering food for grading sessions. I hated this. I had to keep track of the number of entrees, vegetarians, napkins, forks, beverages, etc. And the one time I miscounted the number of forks we'd need, the person who didn't get a fork got upset. It was terrible.

Slide 6



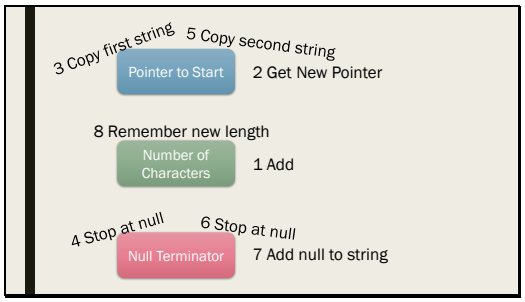
What I didn't learn until later was that I was never supposed to be ordering these items directly from restaurants. Instead, I was supposed to place a catering order directly through some catering company Cornell contracted with, who would take down just a few simple things (# of guests, date, etc.) and then handle all the complexity for me. Notice how much simpler this would have made my life, and how much more time it would've freed up for me to do other things. I shouldn't have had to think about the complexity behind how catering is done – from my perspective, I would've just had one simple catering order. Hold onto that thought.

Slide 7



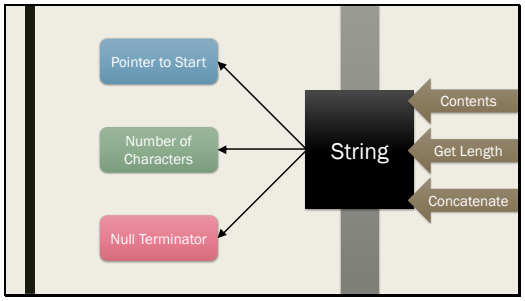
Here's a more interesting story. Back in the olden days, before programming languages had strings, to work with text, you'd have to get a pointer to some memory location, write the text into memory, remember to stick a zero / null byte at the end, and then remember the length of the string yourself.

Slide 8



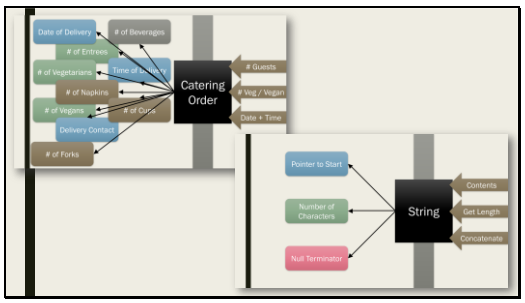
This meant that doing something as simple as concatenating two strings required almost a dozen steps, involving manually copying memory around, doing arithmetic to figure out the length, and manipulating the null terminator bytes.

Slide 9



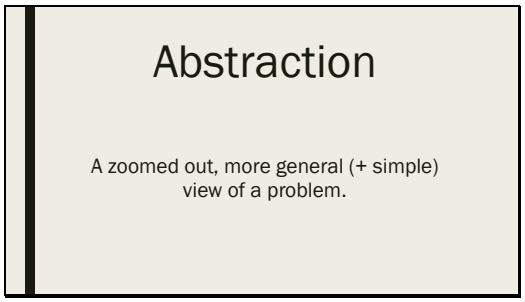
We, of course, know the solution to this problem. We introduced the notion of a "string," which allows us to interface with it in simple ways (telling it to concatenate, for example). And then the string implementation handles all the complexity of the memory for us. We, as users of strings, no longer need to worry about the details of what goes on inside. This made our lives easier.

Slide 10



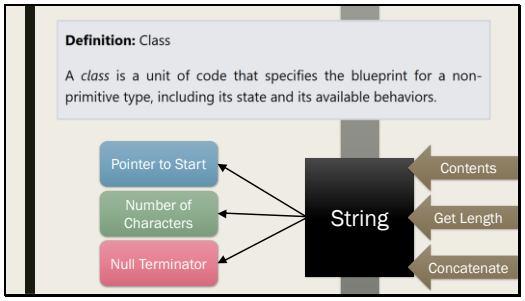
And if you zoom out and squint really hard, hopefully you agree that these are really the exact same story. It's a story of taking a bunch of really complicated, connected pieces of state and simplifying how we deal with it, by turning them into one simpler thing (or "object").

Slide 11



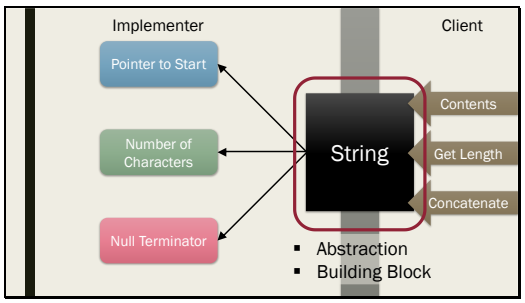
This is the most foundational of computer science principles, the abstraction. The idea that if we zoom out and view something in a more general way, it also makes our lives simpler and easier. It makes code easier to wrangle.

Slide 12



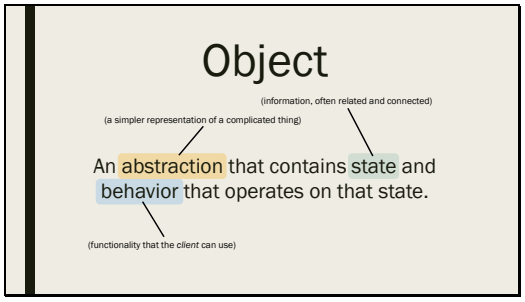
And indeed, look what we have drawn. On the left are complicated pieces of information – or state – and on the right are the things the user might want to do – the behavior – which are exactly the things we saw in the original definition.

Slide 13

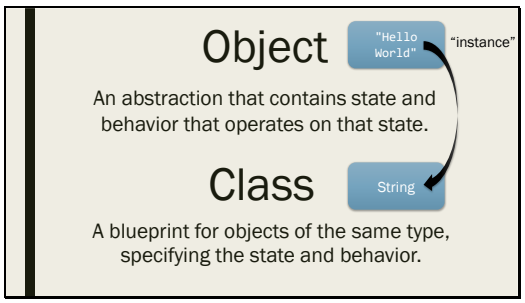


Indeed, here in the middle is what we call an object. It's an abstraction – a simplification of the complexity on the left, built by some implementer, who is giving the client using the object a simpler set of behaviors. The real power is that these abstractions become building blocks for the client. Now that they no longer need to worry about the details, they can view this simpler object as just one thing they can use (think about how you use Strings in your own programs, without ever thinking about the details of memory). This means the client can build even bigger programs than before. And if they turn those into even bigger building blocks, then someone else can use it to build even bigger software. This is how software development is able to scale up – by layering ever more abstractions, no one person is ever straddled with more complexity than they can handle.

Slide 14

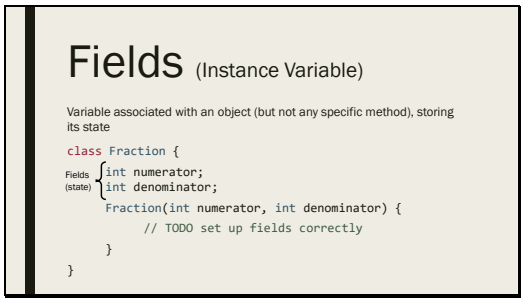


Slide 15

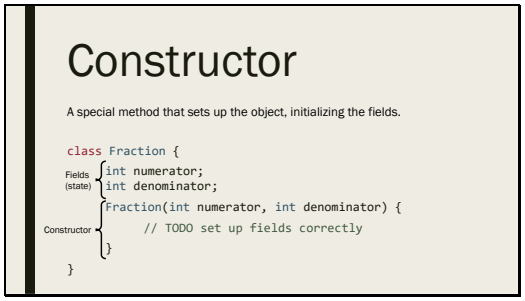


We say that "`Hello World`" is an instance of the `String` class.

Slide 16

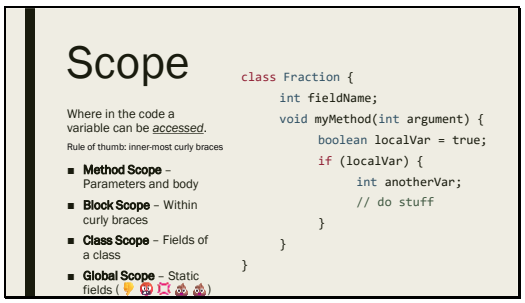


Slide 17



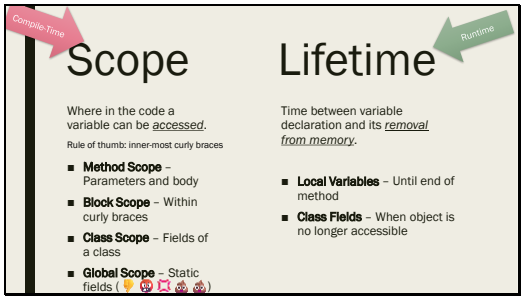
The constructor is a special method, and notice that it does not have a return type. It must be named the exact same thing as the name of the class. Its job is to set up all the fields with the correct values.

Slide 18



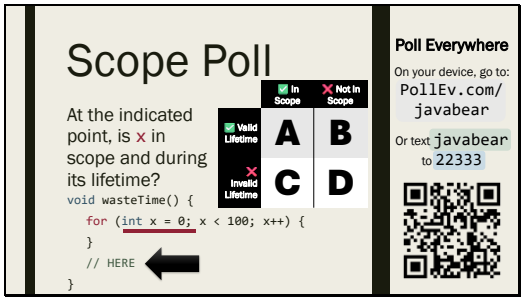
In general, a variable is in scope starting from where it is declared and ending at the closing curly brace of the inner-most curly brace. There are a few exceptions. Method arguments are in scope throughout the entire method, variables declared at the top of for loops are in scope through the end of the for loop, and class fields are in scope throughout the entire class, even before the declaration. There is also global scope, which is bad and we will not use.

Slide 19



Unlike scope, which describes where in your code the compiler allows you to access a variable, lifetime describes when the value of a variable is in memory at runtime. For local variables, this doesn't happen until the end of the method, even if the variable left scope before the method is over.

Slide 20



Correct Answer: B
The variable x is block scoped within the for loop, and so its scope ends at the end of the loop. However, as a local variable, it stays on the stack in the method's call frame until the method returns.

Slide 21

Shadowing

When a variable in an inner scope has the same name as one outside, making the outer one inaccessible.

```
class Fraction {
      int numerator;
      int denominator;
      Fraction(int numerator, int denominator) {
            numerator = numerator;
            denominator = denominator;
      }
}
```

There's an annoying thing that can happen where, if a local variable has the same name as a class field, by default, Java always uses the innermost scoped variable by default. In this example, we want to set the field named `denominator` to the value of the parameter also named `denominator`. However, as written, we cannot access the field.

Slide 22

`this` Keyword

`this` refers to the current object.

```
class Fraction {
      int numerator;
      int denominator;
      Fraction(int numerator, int denominator) {
            this.numerator = numerator;
            this.denominator = denominator;
      }
}
```

The solution is to use the `this` keyword. In Java, `this` always refers to the current object itself.

Slide 23

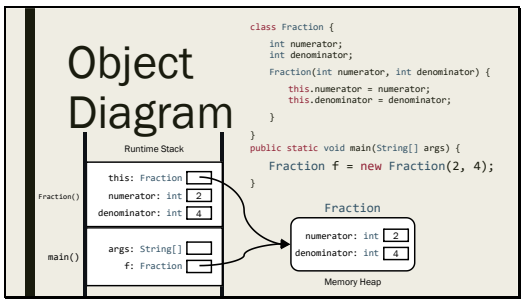
Constructor Demo

```
class Fraction {
      int numerator;
      int denominator;
      Fraction(int numerator, int denominator) {
            this.numerator = numerator;
            this.denominator = denominator;
      }
}
```

Use the constructor to create an object by calling it with the `new` keyword.

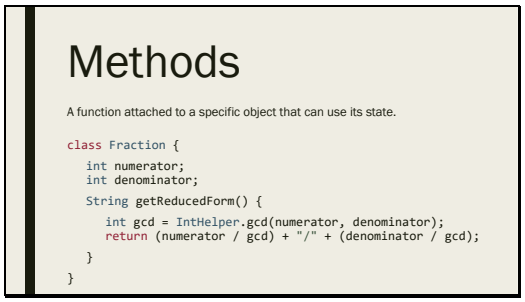
```
Fraction f = new Fraction(2, 4);
```

Slide 24

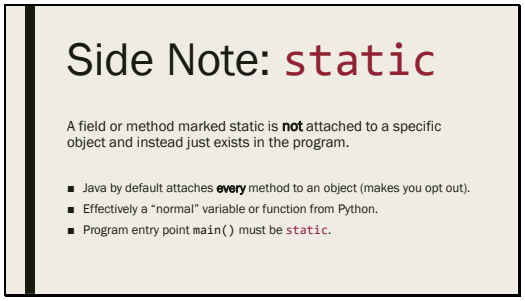


Note that when running the constructor, a pointer to the object itself is available in the stack, named `this`. In effect, every time we write something like `this.numerator`, we are following the arrow from `this` to the object, and then finding the `numerator` field inside.

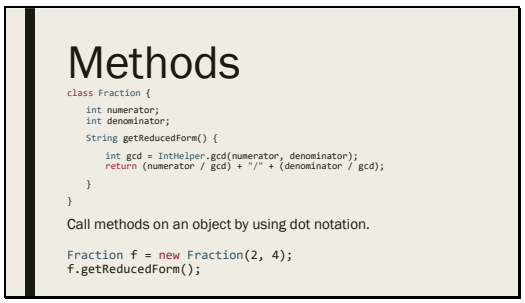
Slide 25



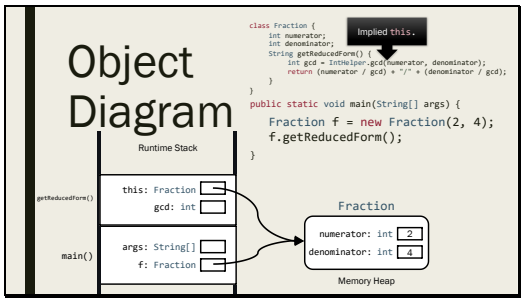
Slide 26



Slide 27

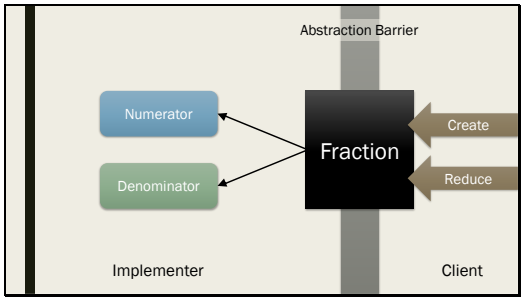


Slide 28



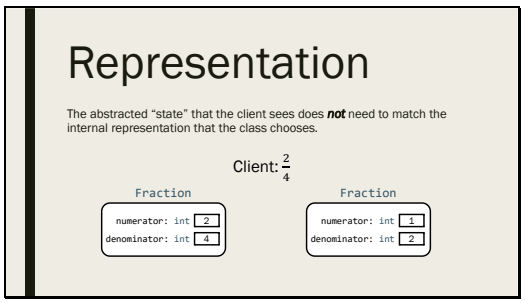
Note inside our implementation of `getReducedForm()` that I have left off the `this` keyword in front of the field names. As a convenience feature, if there is no local variable with the same name, then simply writing the name of a field is enough for Java to figure out that you are referring to the field in the object. There is an implicit `this` in front of the field accesses.

Slide 29



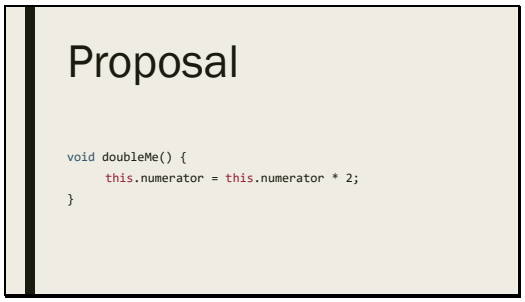
Let's say we want to make the `getReducedForm()` method more efficient by only computing the reduction once. We could choose to instead do the reduction in the constructor itself and store the reduced form directly. This is okay, because from the client's point of view, they don't need to worry about what state is stored inside the object. We can say that there is an "abstraction barrier." What we, as implementers, choose to store inside the object is not something the client wants to think about (much like how I never want to think about purchasing plastic forks again). So we can make different choices here…

Slide 30



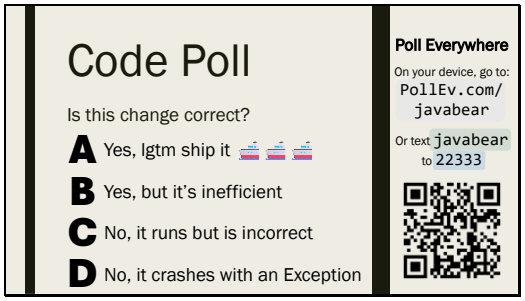
Most notably, the client sees an abstracted state, and not the real values of the variables inside. The client sees that they told us they need a fraction 2/4 reduced, but whether we store their request as 2/4 or ½ is not their problem. So we can simply choose to store the fraction in reduced form upfront, saving time later.

Slide 31



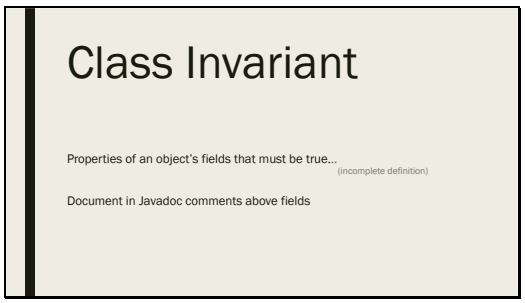
Now let's say we need to double a fraction. Imagine someone writes code that looks like this.

Slide 32



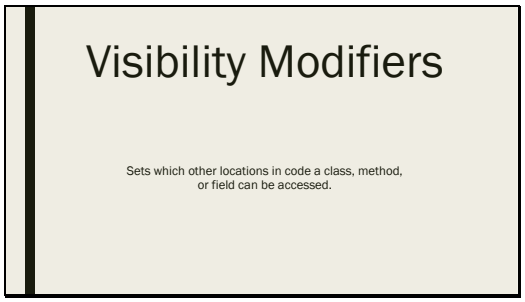
Correct Answer: C
In the code sample shown in class, the `doubleMe()` method had been written to simply double the numerator. However, the `getReducedForm()` method assumes the numerator and denominator fields are always in reduced form, and so is liable to output the incorrect answer if, for example, we start with ¼ and double the numerator.
(lgtm = looks good to me)

Slide 33



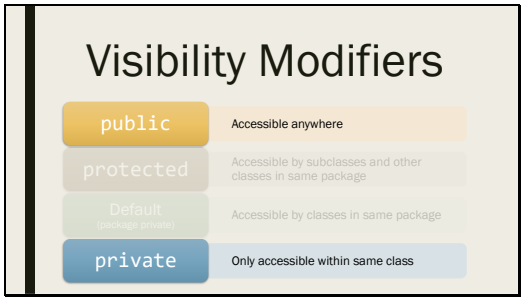
What happened in the previous example is that our code had made an assumption about the state of the object (that the fields were always reduced), and so by breaking that assumption, we broke the class. These assumptions form the class invariant and should be documented above each field they apply to.

Slide 34



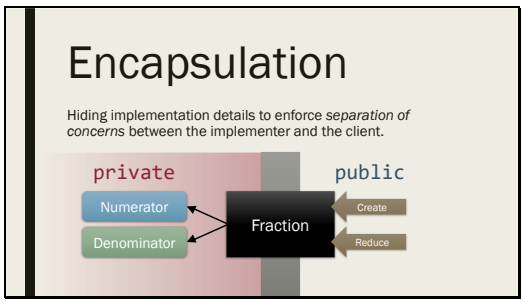
To clearly mark to the client (and the compiler) which methods are things they should use and what is meant to be handled for them, we can use Java's visibility modifiers.

Slide 35



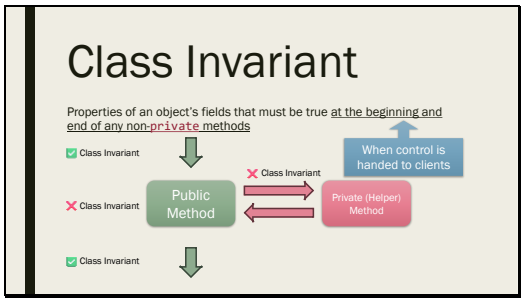
Anything marked `public` is meant to be seen and used by clients of the object. Anything marked `private` is signaling to clients that this is not their concern; the implementer will take care of this for them. The compiler will even help ensure this is true by erroring if the client tries to access something `private`.
There are two others, `protected` which will be discussed next lecture, and the default (when you leave off a visibility modifier) which, going forwards, is practically useless.

Slide 36



The ability to mark things `private` helps us enforce the principle of encapsulation, where details behind the abstraction barrier are hidden from the view of the client.

Slide 37



There's an important caveat on the definition of class invariants, and it's a carve-out for private methods. You can mark methods either public or private, but even though it's just a simple visibility modifier change, they're actually very different things. A public method is part of the object's public behavior exposed to the client. It's one of the main, defining features of the object. A private method, meanwhile, is just a helper function. It's code that helps other, public methods, do their thing. As a result, just like how it's okay for a loop invariant to be false in the middle of a loop iteration, it's okay for the class invariant to be broken during a public method, and also inside a private method called by a public method (which is morally just part of the functionality of the public method). In other words, the class invariant only needs to be true when control is handed back to the client (before and after public methods, including when the constructor finishes).

Slide 38

Tip: Check Invariants

As a debugging tool, consider writing a method that returns whether the class invariant holds and calling it with `assert` when it should.

```
private boolean invariantSatisfied() {
      if (denominator <= 0) { return false; }
      return gcd(numerator, denominator) == 1;
}

// in public methods:
assert invariantSatisfied();
```

As defensive programming, it's a good idea to write a private method that checks if the class invariant is true. This method can return a boolean, and can be called with the `assert` keyword at the end of every public method that mutates the object's state. This way, while debugging, you can be alerted if you accidentally break the invariant yourself, and then, in production, turning off `assert` statements will make the code run faster.

Slide 39

Recap

- Build larger programs by *abstracting* details into simpler building blocks
- Objects help by *encapsulating* inter-connected state and behaviors into a single type
  - Client using object sees simple behavior
  - Implementer of object handles complex state
- Classes define types of objects
  - Contain fields (state) and methods (behavior)
  - Constructor = special method that sets up the object
- Class Invariant specifies rules that the state should follow
- Visibility Modifiers enforce encapsulation by hiding details from users

Slide 40