CSCI 161
Introduction to Computer Science

Est. 1888
UNIVERSITY of
PUGET SOUND

Department of Mathematics
and Computer Science

Lecture 3b
Abstraction & Modularity

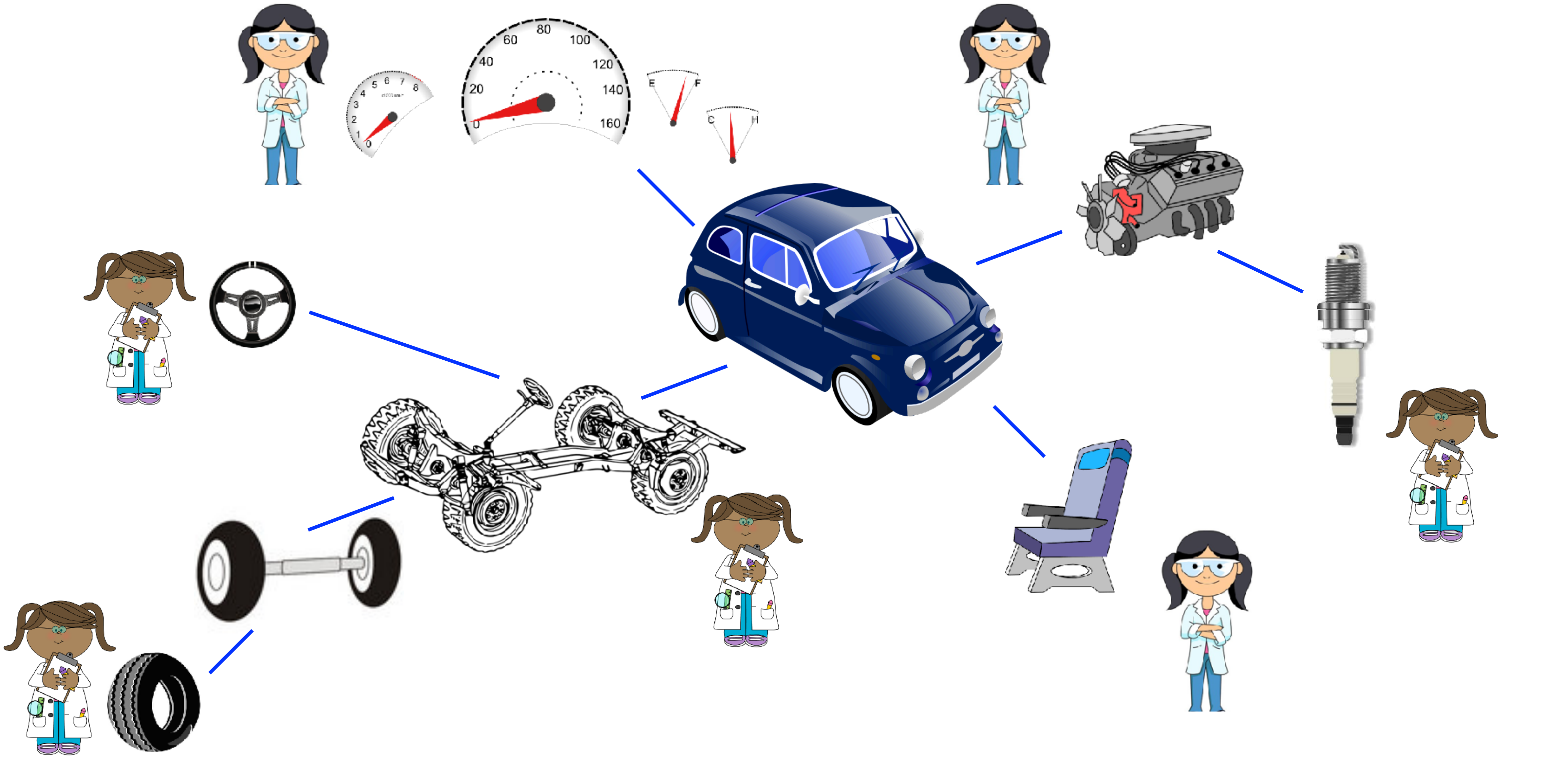
How Do We Build Complex Systems Like Cars?

- Consider making something as complicated as a car
  - Would you start by building layer-by-layer from the ground up?
  - First, take raw aluminum ore and flatten/weld into a frame
  - Second, make tires out of rubber, connect to tire rods and axels next
  - Third, build an engine from scratch (every nut and bolt and chain belt, ...)
  - and so on...

- What are the potential problems with this approach to making cars?

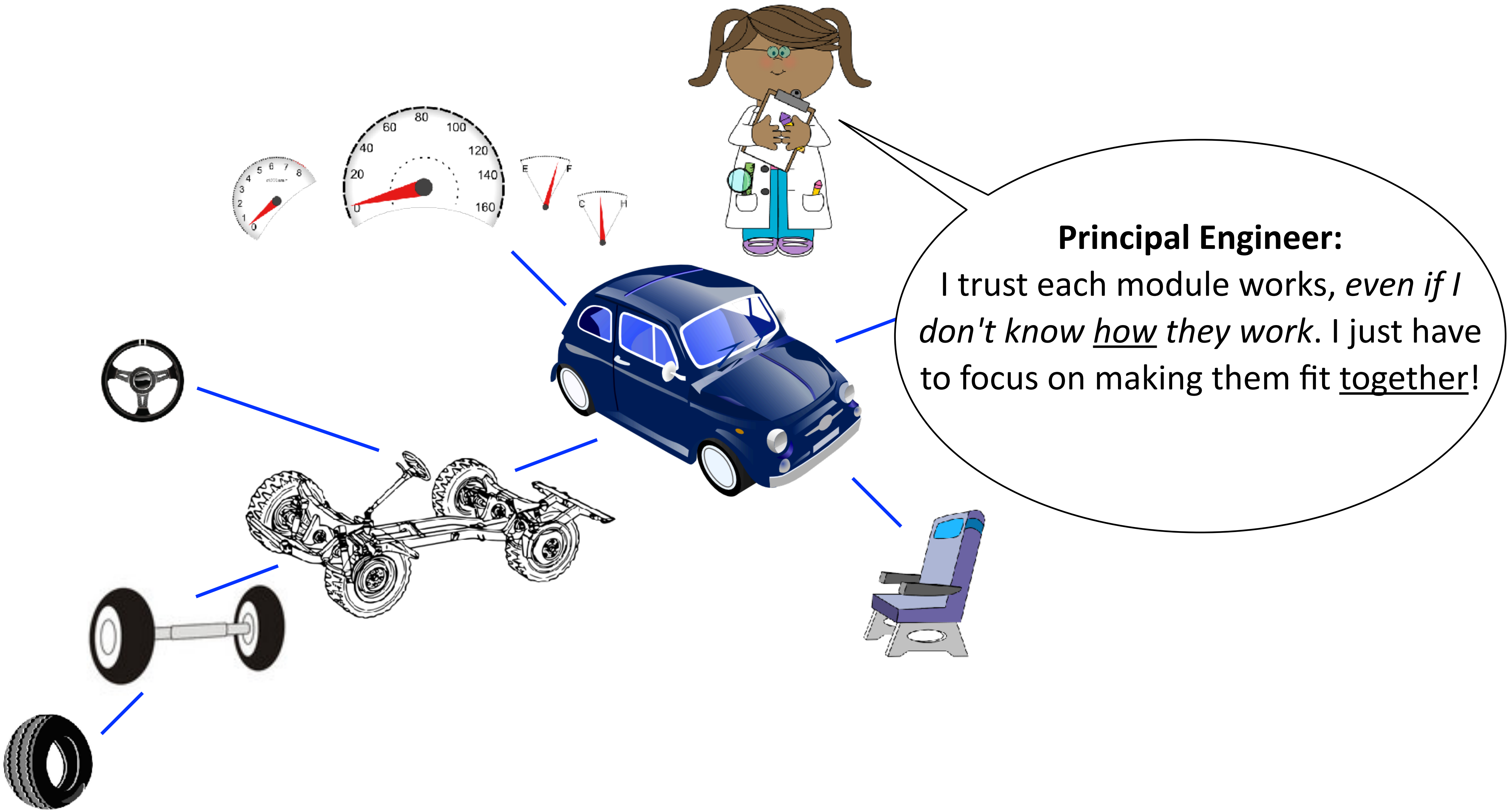
Abstraction and Modularity

- *"Modularity":* Divide the whole into smaller, more manageable pieces (i.e., "modules").



Abstraction and Modularity (Cont.)

- *"Abstraction":* Ignore the inner details of the modules. Trust that they work, and use them to achieve high-level objective.



Principal Engineer:
I trust each module works, *even if I don't know how they work*. I just have to focus on making them fit together!

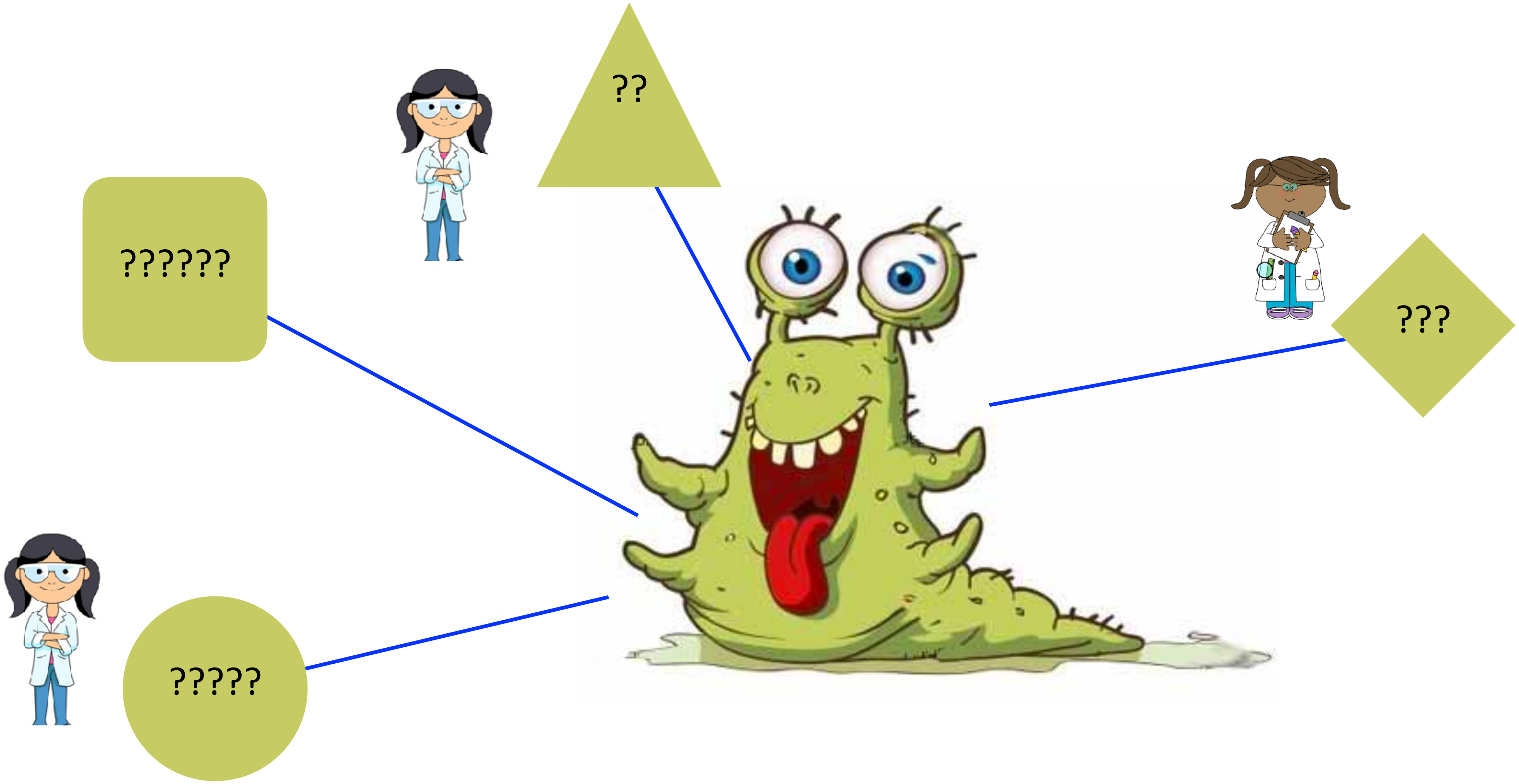
How Do We Build Complex Software Systems?

- For example: an **`Organism`** class
  - Can store and retrieve a thought
  - Can eat and retain food
  - Can digest food
  - Can sleep and wake up

- How much state do I need for this class?
  - Lots of instance variables to keep track of!
    - Amount of food ingested, amount of food digested
    - Awake or not?
    - What is it currently thinking about?
    - And even more!
    - *We could create a big class... or practice abstraction and modularity*

Decomposing the Organism

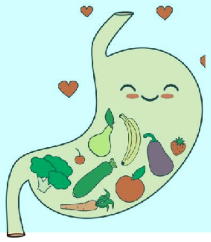
- An **`Organism`**, like a Car, is also complex.
  - How do we *modularize* an organism? **What *are* its independently manageable modules?**



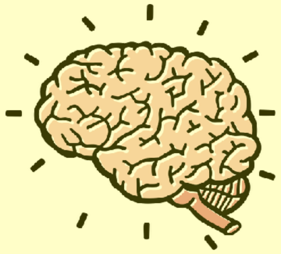
Toward Modularity

- We don't know its modules until we define what an Organism can do.

- Let's say all **Organisms** can...
  - Can eat (and digest) food:
    - `eat()`
  - Can store and retrieve a thought
    - `speak()`, `remember()`
  - Can sleep and wake up
    - `sleep()`, `wakeUp()`



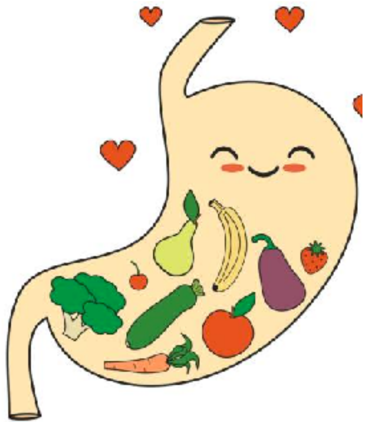
Say... we *have* a **Stomach** class that lets us ingest and digest...



These are similar functions... Create a **Brain** class!

Modularity: Use the Existing Stomach Class

- A **Stomach** keeps track of the amount of food ingested and digested.



- *"Application Programming Interface (API)"*
  - An API is a "user manual" for the class
  - Lists the available constructors and methods

Signature	
`Stomach()`	Creates a new, empty Stomach
`int getAmountFood()`	Returns the amount of food in the stomach
`int getAmountDigested()`	Returns the amount of food digested
`void ingest(int amount)`	Ingests the given amount of food. Ignores negative input.
`void digest()`	Digests a random amount of food in the stomach. (Also removes that amount from stomach).

Modularity: Define a Brain Class

- A brain can...
  - Hold a single *thought*, like "I'm hungry."
  - Keep track of whether it is *asleep*.

- **Ask:** What instance variables does a Brain need?
- **Ask:** What does a Brain know how to do?
  - `setThought()` - Inputs a thought, and stores it in the brain.
  - `getThought()` - Returns the current thought.
  - `setAwake()` - Sets the status of the brain to either awake (true) or asleep
  - `isAwake()` - Returns whether or not the brain is awake

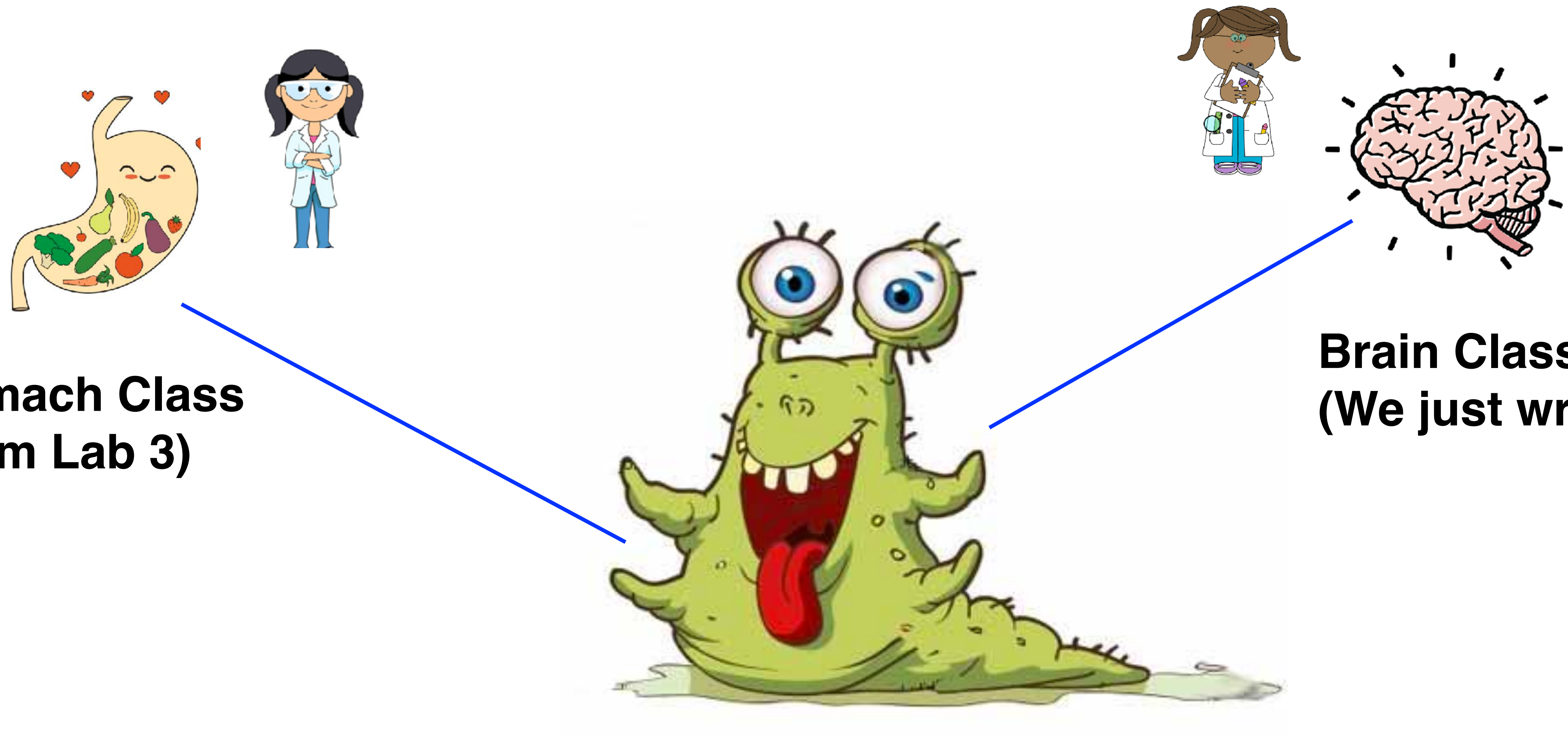
Brain API

- Here's the Brain's API

Signature	
`Brain()`	Creates a new, empty Brain
`void setThought(String newThought)`	Remembers the specified thought.
`String getThought()`	Returns the current thought.
`void setAwake(boolean newStatus)`	Sets the status of the brain to either awake (true) or asleep (false)
`void isAwake()`	Returns whether the brain is awake (true) or asleep (false)

Abstraction!

- Now we can write Organism class without writing brain and stomach functions. Just *use* them!

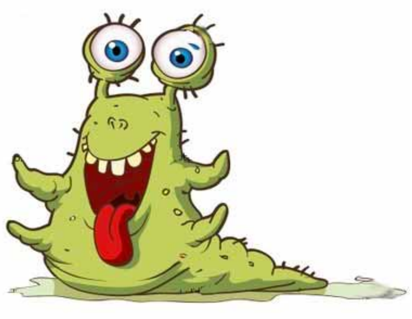


Stomach Class
(From Lab 3)

Brain Class
(We just wrote it)

Organism Class
(Now we can write it!)

Abstraction: Now Write the Organism Class!



- An organism has the following instance variables:
  - A name, a stomach, and a brain
- An organism's methods:
  - `sleep()` - Prints "Zzz" and puts brain to sleep (no action if already sleeping)
  - `wakeUp()` - Prints "Yawn" and wakes the brain up
    - No action if not sleeping
  - `eat()` - Inputs an amount to ingest, prints "Nom nom" to screen, digests too.
    - No action if sleeping
  - `speak()` - Prints the current thought
    - No action if sleeping
  - `remember()` - Inputs a thought and remembers it in the brain. Prints "Interesting..." to the screen.
    - No action if sleeping

Outline

- Data Types
  - Primitives vs. Classes
- Abstraction and Modularity
  - Organism Class
- Useful APIs
  - String
- Conclusion

Review: Object-Oriented Programming

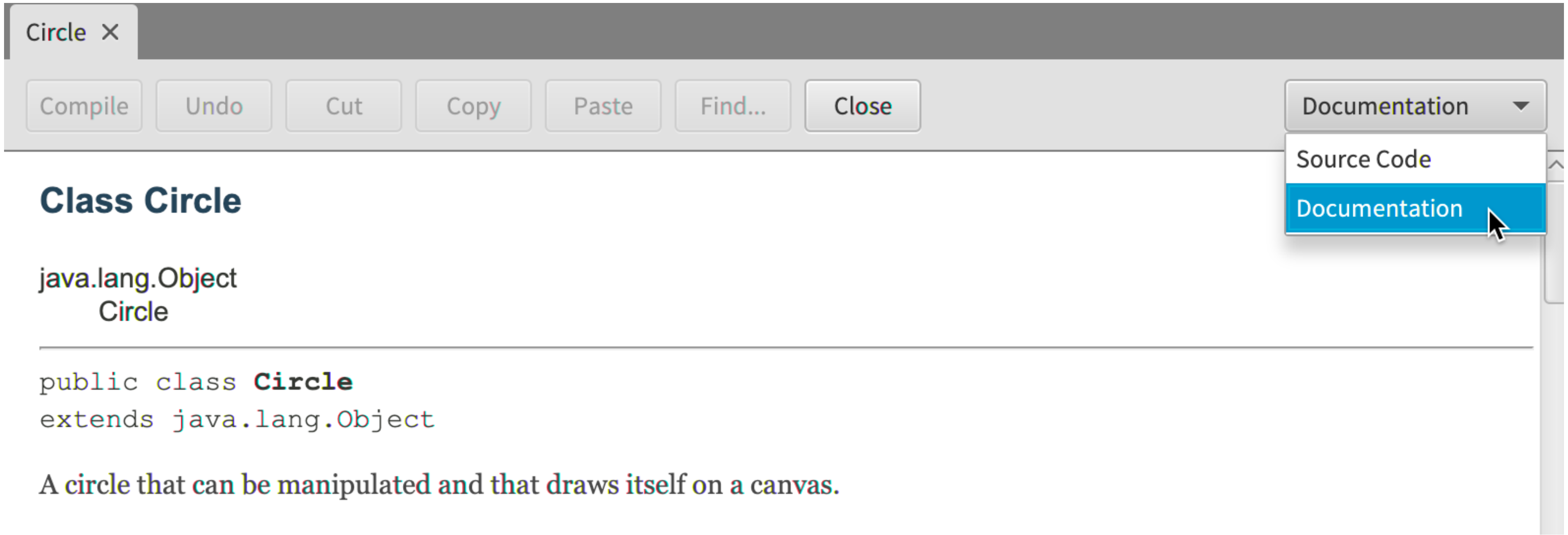
- Modularity
  - Break down a problem into easier-to-tame units
    - Ex: Instead of one monolithic Organism class,
    - Use Brain + Stomach + Limited Organism Code

- Abstraction
  - Use existing classes to help us achieve our goals
    - No need to know nitty-gritty of the other classes
    - Just need to know how to leverage them
  - The instruction manual for other classes is known as their "Application Programming Interface"

Accessing APIs

- Fortunately, lots of ways to access APIs!

- If the class is given in your BlueJ Project folder...
  - Double-click on the class to open code editor
  - Top-right corner, select "Documentation"



Class Circle

java.lang.Object
Circle

```
public class Circle
extends java.lang.Object
```

A circle that can be manipulated and that draws itself on a canvas.

Accessing APIs (2)

- If the class is outside of your project and imported, a simple google search should pull it up.

- For instance, search the web for:
  - "java String api"
  - "java ArrayList api"

- Make sure it takes you to the oracle.com site
  - (Theirs is the most up-to-date)

Outline

- Data Types
  - Primitives vs. Classes
- Abstraction and Modularity
  - Organism Class
- Useful APIs
  - String
- Conclusion

Exploring Strings

- Speaking of *abstraction*, one of the classes/objects you've been using the *whole semester* are **`Strings`**.

- **`Strings`** are objects that represent a sequence of **chars**:
  - Recall that a **char** is a primitive data type, that can hold a single symbol.
  - Each character in the string corresponds to a position (or address, or index)

- A String **`"Hello World!\n"`** is represented in the machine as:

	H	e	l	l	o		W	o	r	l	d	!	\n
Addr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

Strings Are Objects?

- Yes, **`String`** is a class in Java!
  - Each string also has access to various methods.

- Then **`Strings`** must have a constructor? Yes they do!
  - We've never seen the **`new`** keyword being used to instantiate strings.
  - Java hides it from us though.

```
String message = new String("Hello World!");   // This works!
```

```
String message = "Hello World";   // But this syntax is more convenient!
```

Strings Are Special Objects

- Strings are the most commonly used objects in Java.
  - Java had to make them super convenient to use!

- **#1:** Strings have a short-hand constructor
  - (Use of the **new** keyword is not necessary for creating string objects)

```
String message = "Hello World";   // This construction syntax is convenient!
```

- **#2:** Strings have their own handy-dandy operator: **+**
  - To "concatenate" two strings

```
"Hello" + " " + "World"
```

Are Strings "Mutable?"

- We've been calling methods on objects to change (mutate) their state.
    - For instance:

```
Fraction f = new Fraction(3,5);
f.negate();    // changes it to -3/5
f.inverse();   // changes it to -5/3
```

```
Triangle t = new Triangle();
t.changeColor("blue");
t.moveUp();
t.moveLeft();
t.slowMoveVertical(-40);
```

- How about Strings?

```
String str = "hello";
str.toUpperCase();
System.out.println(str);   // what will this print?
```

Immutability of Strings

- #3 Strings are immutable objects.
  - Calling methods on them do not change their internal state at all!

- Then what good are their methods??
  - They can return a new string, though.

- In the previous example, how do you capture the upper-case version?
  - (You need capture or re-capture its return value)

```java
String str = "hello";
str = str.toUpperCase();   // re-capture the upper-case version in str
System.out.println(str);   // This prints HELLO
```

The String API (Selected Methods)

Length

`public boolean isEmpty()`	Returns true if and only if the `length()` is zero.
`public int length()`	Gets the length of the String.

Comparison

`public int compareTo(String other)`	Returns 0 if strings are equal, -1 if current string is "less than" input; positive value if current string is "greater than" input.
`public boolean equals(String other)`	Tests if two strings are equal. Case sensitive.

Extraction

`public char charAt(int pos)`	Returns the character at the given position, `pos`.
`public String[] split(String delimiter)`	Splits the string into substrings around the given delimiter.
`public String substring(int begin)`	Returns a copy of the String starting from position `begin` to the end.
`public String substring(int begin, int end)`	Returns a copy of the String starting from position `begin`, ending at position `end – 1`.

The String API (2)

Manipulation

`public String toLowerCase()`	Returns a copy of the String in lower case.
`public String toUpperCase()`	Returns a copy of the String in upper case.
`public String trim()`	Returns a copy of the String omitting any leading and trailing spaces.

Search and Replace

`public int indexOf(String str)`	Returns starting position of str if found, or -1 if not found in the current string
`public String replace(String key, String rep)`	Returns a copy of the String after replacing all occurrences of key with rep

Examples using String Methods

- Getting the length of a **`String`**
  - This method is widely used

```
String school = "University of Puget Sound";
int size = school.length(); // size gets 25
```

- Search and Replace (case-sensitive)

```
String name = "Adam A. Smith";
String shortenedName = name.replace("A. ", "");
System.out.println(shortenedName);  //Adam Smith
```

Extraction Examples

- Extracting a Substring
  - Keep in mind that Java subtracts 1 from the given **end** index.

- Example: Extract first name:

```
String fullname = "Brad Richards";
String firstname = fullname.substring(0, 4);
```

- Example: Extract last name:

```
String fullname = "Brad Richards";
String lastname = fullname.substring(5, fullname.length());
```

Example: Create a class, StringExercise

- Puget Sound email addresses are formed using the first initial appended to the last name appended to @pugetsound.edu

- **Write** an email creation method called **`createEmail()`**:
  - Two input parameters: first name & last name
  - Returns a Puget Sound email address in lowercase
    - But if *either* input is an **empty string ("")**, return an **empty string**.

- Example Usage:

```
String myEmail = createEmail("David", "Chiu");
System.out.println(myEmail); // outputs "dchiu@pugetsound.edu"
```

```
String myEmail = createEmail("David", "");
System.out.println(myEmail); // outputs ""
```

Email Solution

```java
/**
 * Creates a puget sound email
 * @param first The first name of student
 * @param last  The last name of student
 * @return the email address, or empty string if either name is not given
 */
public String createEmail(String first, String last) {
    if (first == null || last == null || first.isEmpty() || last.isEmpty()) {
       // one or both inputs were empty
       return "";
    }

    // convert both to lower case
    first = first.toLowerCase();
    last = last.toLowerCase();
    return first.charAt(0) + last + "@pugetsound.edu";
}
```

Your Turn: Pig-Latin-fying Words

- Write a method **pigLatin**(String word) that inputs a word and returns the Pig Latin version of the word.
  - If a word starts with a consonant, swap that letter to the back, hyphenate, and concatenate **"ay"** to it.
  - If a word starts with a vowel, just concatenate **"-way"** to that word

```
System.out.println(pigLatin("hello"));
> ello-hay

System.out.println(pigLatin("Mice"));
> ice-May

System.out.println(pigLatin("circle"));
> ircle-cay

System.out.println(pigLatin("apple"));
> apple-way

System.out.println(pigLatin(""));
>
```

Pig Latin Solution

```java
/**
 * Pig Latinfies a word.
 * @param word
 * @return the pig-latin version of the specified word
 */

public String pigLatin(String word) {
    if (word == null || word.isEmpty()) {
        return "";
    }

    if (isVowel(word.charAt(0))) {
        // first letter is a vowel
        return word + "-way";
    }
    // first letter is a consonant
    return word.substring(1) + "-" + word.charAt(0) + "ay";
}


/**
 * @return true if the given character is a vowel; false otherwise
 */
private boolean isVowel(char c) {
    return (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
}
```

Your Turn!

- Write a method called vowelsAtEnds:
  - Inputs a word (String)
  - Returns **true** if word starts *and* ends with vowel; **false** otherwise
    - Assume standard vowels only: a,e,i,o,u
    - Don't assume word will be given in lower case

- Example Usage:

```
System.out.println(vowelsAtEnds("Ada"));             // true
System.out.println(vowelsAtEnds("ice cream"));       // false
System.out.println(vowelsAtEnds("  UMBRELLA   "));   // true (oooh, spaces)
System.out.println(vowelsAtEnds(""));                // false
```

Solution

```java
/**
 * Tests whether a string starts and ends with a vowel.
 *
 * @param  s Some given string to test
 * @return true if the given string starts and ends with a vowel
 *         false otherwise, or if string is empty.
 */
public boolean vowelsAtEnds(String str) {
    if (str == null || str.isEmpty()) {
        return false;
    }
    // trim leading and trailing spaces and convert to lower case
    str = str.trim().toLowerCase();
    return isVowel(str.charAt(0)) && isVowel(str.charAt(str.length()-1));
}

/**
 * @return true if the given character is a vowel; false otherwise
 */
private boolean isVowel(char c) {
    return (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
}
```

Conclusion

- Abstraction is divide and conquer in software
  - Break up big problem into small, manageable pieces
  - Make sure you do a good job programming those pieces
  - Orchestrate together later to solve bigger problem
  - One of the important concepts in CS

- We also saw primitive types and their operators
  - What about object types? What are their operators? *(Next)*