Outline

- Data Types
  - Primitives vs. Classes

- Abstraction and Modularity
  - Organism Class
  - Clock Class

- Useful APIs
  - String
  - Random

- Conclusion
How Would You Build This?
How Would You Build This?

- An organism:
  - Can store and retrieve a thought
  - Can eat and retain food
  - Can digest food
  - Can sleep and wake up

- How much state and methods do I need for this class!?

- Clue: Think like a lego builder!
  - Build things piecemeal
Abstraction and Modularity

- It's tempting to build a single `Organism` class
  - After all, it's worked well for us so far...
  - Fine for trivial classes like `TicketMachine`, and `Calculator`
    - They just need to store a couple numbers
    - Methods operate directly over those numbers

- But more complicated classes like `Organism` have way more moving parts:
  - Analogy: How do factories engineer a new car, or a jet, or a ship, or an organism?
Abstraction and Modularity (Cont.)

- Brute-force method:
  - Pre-req: One expert knows how to make *everything* that goes into a car
  - First, take raw aluminum ore and flatten/weld into a frame
  - Second, make tires out of rubber, connect to tire rods (also crafted)
  - Third build engine from scratch (every nut and bolt and belt, ...)
  - Then....

- Result: Giant nope.
  - Low throughput
  - Probably error prone
  - Future maintenance and production of the car
Modularity: Divide the whole into well-defined parts, which can be built separately and interact in well-defined ways.

• In other words, "divide and conquer"
Abstraction and Modularity (Cont.)

- **Abstraction**: Ignore details of each part, focus on the high-level task

  I'm confident each component works, *even if I don't know how each work*. I just have to focus on making them work together!
Back to the Organism then...

An organism:

- Can eat and retain food:
  - eat()
- Can store and retrieve a thought
  - think(), remember()
- Can sleep and wake up
  - sleep(), wakeUp()

These are similar functions... create a **Stomach** class!

These are similar functions... create a **Brain** class that deals only with those...
Let's Modularize an Organism

- First, let's remind ourselves what a Stomach can do
  - Hand out Stomach Application Programming Interface (API)
An application programming interface (API) lists:

- The available constructors
- The available methods

You should be able to have the API and know everything you need to know on how to use the class.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public Stomach()</td>
<td>Creates a new, empty Stomach</td>
</tr>
<tr>
<td>public int getAmountFood()</td>
<td>Returns the amount of food in the stomach</td>
</tr>
<tr>
<td>public int getAmountDigested()</td>
<td>Returns the amount of food digested</td>
</tr>
<tr>
<td>public void ingest(int amount)</td>
<td>Ingests the given amount of food. Ignores negative input.</td>
</tr>
<tr>
<td>public void digest()</td>
<td>Digests a random amount of food in the stomach. (Also removes that amount from stomach).</td>
</tr>
</tbody>
</table>
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 fields does the 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 (false)
  - *getAwake* - Returns whether or not the brain is awake
Here's the Brain's API

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public Brain()</code></td>
<td>Creates a new, empty Brain</td>
</tr>
<tr>
<td><code>public void setThought(String newThought)</code></td>
<td>Remembers the specified thought.</td>
</tr>
<tr>
<td><code>public String getThought()</code></td>
<td>Returns the current thought.</td>
</tr>
<tr>
<td><code>public void setAwake(boolean newStatus)</code></td>
<td>Sets the status of the brain to either awake (true) or asleep (false)</td>
</tr>
</tbody>
</table>

**Brain API**
Now, Exploit Abstraction!!!

- We are sure both Stomach and Brain classes are working.
  - Don't worry about details of *how they work*.
  - Complete the Organism class!

I'm confident each component works, even if I don't know how each work. I just have to focus on making them work together!
Abstraction: Now Write the Organism Class!

- An organism has the following fields:
  - 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 brain up
    - No action if not sleeping
  - `eat` - Inputs an amount to ingest, prints "Nom nom" to screen, digests too.
    - No action if sleeping
  - `think` - 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
  - Clock Class

- **Useful APIs**
  - String
  - Random

- **Conclusion**
How Would You Build This?

- A digital clock program
  - Assume it displays 24-hour military time
  - Clock can display hour and minute only
Abstraction and modularity are crucial skills for computer scientists

- Some software applications are as complex as cars and airplanes

Need to start developing these skills. We'll start small.

- But first... is modularity needed?
Is Modularity Needed for Clock?

- Are we sure this clock is complicated enough to need modularity?
  - Clock remembers two numbers: hours and minutes
    - But need to ensure proper ranges for each
      - 0-23 for hours, 0-59 for minutes
  - Need to make sure single-digit numbers are always displayed using 2-digits
    - 8 gets displayed as "08"
  - The hour and minute-number can both tick up
    - Might wrap around!

- Easier if we could just ignore the number display's inner workings and focus on the inner working of a clock!
Dividing and Conquering a Digital Clock

- **Modularize** *(Divide)*:
  - What are the well-defined sub-components of a clock?
  - Code each sub-component separately as a class

A NumberDisplay object has a **value** and a **limit**. The **value** can be incremented, but wraps around when it reaches **limit**.

**Two instances of "2-digit number displays"**

A ClockDisplay has two NumberDisplays: an hour and a minute. It can read each NumberDisplay and increment them.
Step 1: Determine the fields

- What does every number display need to remember?
  - The **number** it's displaying (What's the type?)
  - The **limit** before wrapping around (What's the type?)
    - For minutes, it's 60
    - For hours, it's 24
    - For a car's odometer, it's 999999
    - For some numbers like (*e.g.*, a score or someone's height), there is no wraparound limit (use 0)

- *Let's start writing the class (BlueJ)*
NumberDisplay Class (Constructors)

- **Step 2: Constructors**
  - They give the fields we just defined their initial values
  - Always have a default constructor
    - Requires no input from user. Sets number display to 0, with no wraparound limit (0).
  - Another one that lets user input the starting the wraparound limit.
    - Also sets number display to 0

- **Update the class (BlueJ)**
Step 3: Methods

- What can NumberDisplays do?
  - Getter for the number value: getValue
    - Return type? Any params?
  - Setter for the number value: setValue
    - Return type? Any params?
  - Return the number displayed in pretty 2-digit format (e.g., 7 becomes "07"): toString
    - Return type? Any params?
  - Tick up the number (might cause it to wrap around): tick()
    - Return type? Any params?

Update the class (BlueJ)
Testing NumberDisplay

- From code pad

```java
NumberDisplay nd = new NumberDisplay(12);
System.out.println(nd.toString());
> "00"  (String)

nd.tick();
nd.tick();
System.out.println(nd.toString());
> "02"  (String)

nd.setValue(11);
System.out.println(nd.toString());
> "11"  (String)
```
Testing NumberDisplay

From code pad

```java
NumberDisplay nd = new NumberDisplay(12);
System.out.println(nd.toString());
> "00"   (String)

nd.tick();
nd.tick();
System.out.println(nd.toString());
> "02"   (String)

nd.setValue(11);
System.out.println(nd.toString());
> "11"   (String)

nd.tick();
System.out.println(nd.toString());
> "00"   (String)
```
ClockDisplay Class

- Beauty of Abstraction (*Conquering Step*):
  - Now we can focus on the task of building a clock
    - Without worrying about:
      - How will the numbers will wrap around?
      - How will the numbers be displayed in 2-digit format?
      - What if clock tries to set an invalid hour/minute?

- Step 1: Determine the fields
  - Hour display (type?)
  - Minute display (type?)

- *Update the class (BlueJ)*
ClockDisplay Class (Constructors)

- Step 2: Define the constructors
  - Default constructor:
    - Hour and minute start at 0
  - Overloaded constructor:
    - Let the caller choose hour and minute

- Update the class (BlueJ)
Step 3: Define the Methods

- Tick up one minute in the clock
  - Needs to increase hour too sometimes!
- Set time
  - Ask user to specify hour and minute
- Get time
  - Gets the current time in "HH:MM" format

Update the class (BlueJ)
Testing Clock

- From code pad

```java
Clock my_clock = new Clock(23, 58);
my_clock.getTime()
> "23:58" (String)

my_clock.timeTick();
my_clock.getTime()
> "23:59" (String)

my_clock.timeTick();
my_clock.getTime()
> "00:00" (String)
```
Aside: Object Diagram vs. Class Diagram

Object Diagram

Class Diagram
Outline

- Data Types
  - Primitives vs. Classes

- Abstraction and Modularity
  - Organism Class
  - Clock Class

- Useful APIs
  - String
  - Random

- Conclusion
Java's Random Class

- Saw this in Guessing Game Lab
  - Need to `import` the class first
    ```java
    import java.util.Random;
    ```

- Learn how to use it just by reading its *application programming interface (API)*
  - [https://docs.oracle.com/javase/7/docs/api/java/util/Random.html](https://docs.oracle.com/javase/7/docs/api/java/util/Random.html)

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public Random()</code></td>
<td>Creates a new random number generator</td>
</tr>
<tr>
<td><code>public boolean nextBoolean()</code></td>
<td>Returns a random boolean value (50% probability)</td>
</tr>
<tr>
<td><code>public double nextDouble()</code></td>
<td>Returns a random value between 0.0 (incl) and 1.0 (excl)</td>
</tr>
<tr>
<td><code>public int nextInt()</code></td>
<td>Returns a random integer between 0 (incl) and 2^32 (excl)</td>
</tr>
<tr>
<td><code>public int nextInt(int n)</code></td>
<td>Returns a random integer between 0 (incl) to n (excl)</td>
</tr>
</tbody>
</table>
# Java's String Class (Selected Methods)

## String API

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public String()</code></td>
<td>Creates a String object with an empty character sequence. (Wait! Strings have constructors?)</td>
</tr>
<tr>
<td><code>public String(String original)</code></td>
<td>Creates a String object with given character sequence.</td>
</tr>
<tr>
<td><code>public boolean equals(String other)</code></td>
<td>Checks content equality with given String</td>
</tr>
<tr>
<td><code>public boolean equalsIgnoreCase(String other)</code></td>
<td>Same as above, but case-insensitive</td>
</tr>
<tr>
<td><code>public int indexOf(String str)</code></td>
<td>Returns starting position of <code>str</code> if found, or <code>-1</code> if not found</td>
</tr>
<tr>
<td><code>public int length()</code></td>
<td>Gets the length of the String</td>
</tr>
<tr>
<td><code>public String replace(String target, String rep)</code></td>
<td>Returns this String resulting from replacing <code>target</code> with <code>rep</code></td>
</tr>
<tr>
<td><code>public String substring(int begin, int end)</code></td>
<td>Returns this String starting from position <code>begin</code>, ending at position <code>end-1</code></td>
</tr>
<tr>
<td><code>public String toLowerCase()</code></td>
<td>Returns this String in lower case</td>
</tr>
<tr>
<td><code>public String toUpperCase()</code></td>
<td>Returns this String in upper case</td>
</tr>
</tbody>
</table>

---

The `Puget Sound` string has a length of 11, and the character at position 5 is `u`.
Examples using String Methods

- **Really important** to remember: Strings are *immutable* (???)

- Getting length

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

- Find and replace (case-sensitive)

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

- All methods can be "chained" using dot notation (order left to right)

```java
String name = "Krusty";
int len = name.replace("Kr", "R").length(); //5
```
Examples using String Methods (Cont.)

- Extracting a Substring

```java
String fullname = "Brad Richards"
String firstname = fullname.substring(0, 5);
```

- Put Brad's last name inside lastname:

```java
String fullname = "Brad Richards"
String lastname = fullname.substring(???, ???);
```

- Useful exercise: repeat above, but assume someone inputs the full name, and the program doesn't know what's inside (could be Adam Smith, America Chambers, Brad Richards, ...)
More String Exercises

- Write a method `sameLastNames` that inputs two names (first and last), and determines whether their last names are the same.

  Example:

  ```java
  String name1 = "David Lynch";
  String name2 = "Marshawn Lynch";

  if (sameLastNames(name1, name2)) {
    System.out.println("Yes, Same last names!");
  }
  else {
    System.out.println("Different last names!");
  }
  
  public boolean sameLastNames(String name, String anotherName) {
    //TODO: extract both lastnames
    //TODO: compare them for content equality, and return results of this comparison
  }
  ```
Write a method `sameLastNames` that inputs two names (first and last), and determines whether their last names are the same.

**Example:**

```java
String name1 = "David Lynch";
String name2 = "Marshawn Lynch";

if (sameLastNames(name1, name2)) {
    System.out.println("Yes, Same last names!");
} else {
    System.out.println("Different last names!");
}
```

**Solution:**

```java
public boolean sameLastNames(String name, String anotherName) {
    //extract both lastnames
    String lastName1 = name.substring(name.indexOf(" "), name.length());
    String lastName2 = anotherName.substring(anotherName.indexOf(" "), anotherName.length());
    return lastName1.equals(lastName2); //apply string content-equivalence
}
```
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)
Administrivia 2/25

- Hwk 4 posted
  - Due 3/6 (Wednesday)

- Midterm I:
  - Review Wednesday (Bring questions!)
  - Exam Thursday
    - 1 page cheatsheet okay
    - Scratch paper okay
    - Use pencil not pen
Administrivia 2/25 (cont)

Talk today:
- Jared Jackson, Riot Games
- Achievement Unlocked: An Inside Look at the Gaming Industry
- TH 395, 4pm

Last time...
- Object variables vs. Primitive variables
- Reference equality
- Today: content equality, abstraction and modularity