WEEK FIFTEEN

Acknowledgements: Slides created based off material provided by Dr. Michael Raymer and Dr. Travis Doom

RECURSION

- **Recursive method:** any method that calls itself
- Starts with a complicated problem and breaks it down into smaller and smaller problems
  - aka divide and conquer decomposition
- Why use recursion?
  - Sometimes simpler/easier to understand and maintain
  - Sometimes more flexible
  - Sometimes faster
  - **Short answer:** it's another tool in your toolbox

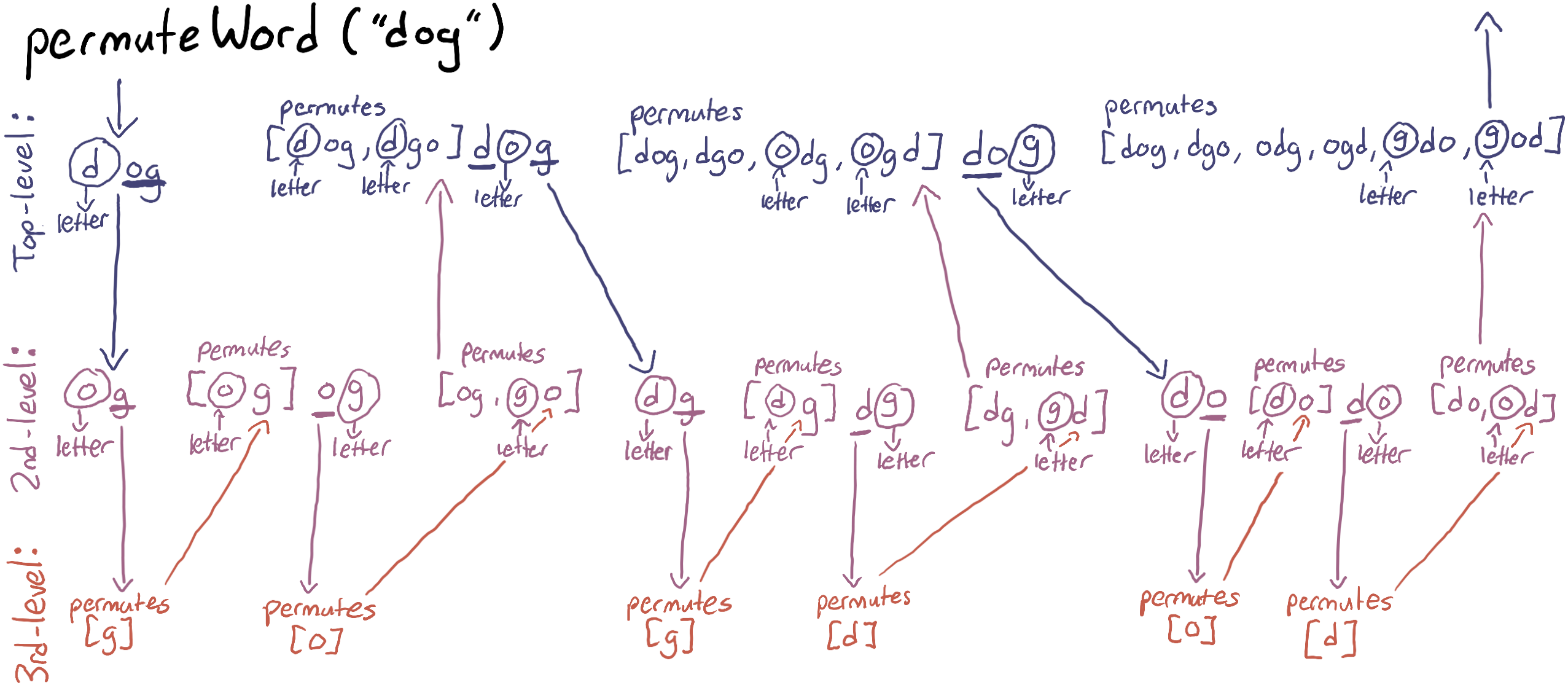
THE BASE CASE

- A condition where we no longer make a recursive call
- Returns a basic and simple value
- Designed to reverse the recursion process to return to the original recursive call

```
public static int sum(String num){
      if (num.length() == 1) {
            return Integer.parseInt(num);
      } else {
            return Integer.parseInt(num.substring(0, 1)) + sum(num.substring(1));
      }
}
```

HOW TO UNDERSTAND RECURSION

- **A few very simple base cases (often just one)**
- **Rules to break down complex cases into simpler cases of the same general problem**
- **Are we done yet? If so, return the results.**
- **If not, simplify the problem (move towards the base case), by constructing a solution from smaller similar problems**
- **If you have an already solved similar but simpler problem, how can that help you solve a more complex problem?**


permuteWord ("dog")
Top-level:
2nd-level:
3rd-level:
letter
permutes
[dog, dgo, odg, ogd, gdo, god]
[dg, gd]
[do, od]
[g]
[o]
[d]

RECURSION PRACTICE #1

- Given a string, compute recursively (no loops) the number of times lowercase “hi” appears in a String
- Example:
  - **IN:** “xxhixx” **OUT:** 1
  - **IN:** “xhixhix” **OUT:** 2

RECURSION PRACTICE #2

- Given a number n, compute the n^{th} Fibonacci number
- The first two numbers in the Fibonacci sequence are 1 and 1
- Each subsequent number is the sum of the two previous numbers
- So, the next number is 1 + 1 = **2**, then 1 + 2 = **3**
- Example:
  - **IN:** 1 **OUT:** 1
  - **IN:** 4 **OUT:** 3
  - **IN:** 8 **OUT:** 21

RECURSION PRACTICE #3

- Merge sort
  - Usually done recursively
  - Divide array into two halves
  - As base case, sort the two remaining values
  - Upon exiting the recursive call, poll items from each half in order

GENERICS

- Allow classes to work with various data types
- Signified by a single capital letter (usually E for Element)
  - Others include N for Number, T for Type, K for Key, V for Value
- Often accompanied by <>
- Help catch typing errors early
- Control the type a container will hold
- Avoid extra typecasting

GENERIC CLASSES

- If we want our class to work with any data type, we add the *type parameter* to the class header inside <>
  - `public class BingoMachine<E>`
- Represents a class with a collection of elements that are stored and procured randomly
- Can store different kinds of objects because of the generic type E

BOUNDED TYPE PARAMETERS

- We use the `extends` keyword to denote the class that restricts our generic type
  - `public class BingoMachine <E extends Number>`
- Now we can only store subclasses of Number in our BingoMachine
- The `extends` keyword is not only used for regular classes but abstract classes and interfaces too
  - `public class BingoMachine <E extends Comparable<E>>`
- Says that the type E must be comparable to itself
- In other words, forces the class E to implement the compareTo method
  - `public boolean compareTo(E otherItem)`

MORE ON GENERICS

- Suppose my BingoMachine class is defined as follows
  - `public class BingoMachine <E extends Number>`
- Now suppose I want to make a method that lets me add an entire ArrayList to my object
  - `public void addAll(ArrayList<E> listToAdd)`
- Would this work with an ArrayList of Number?
  - Yes!
- How about an ArrayList of Double?
  - Double is a child of the Number class
  - BUT, ArrayList<Double> is *not* a subclass of ArrayList<Number>

WILDCARDS!

- Represented by a question mark (?)
- By itself, represents an unknown type (any type)
  - Useful for methods that only require functionality of the `Object` type
  - Or if the code uses methods not dependent on the type parameter
- Can be upper-bounded by adding the `extends` keyword
  - `public void addAll(ArrayList<? extends E> listToAdd)`
  - Accepts any ArrayList of type E
  - AND any ArrayList of a type that is a child of E

LOWER-BOUNDED WILDCARDS

- Suppose I want to be able to add everything from my BingoMachine to an ArrayList
  - `public void addToOtherList(ArrayList<E> otherList)`
- Would this work with an ArrayList of Number?
  - Yes!
- What about an ArrayList of Object?
  - Number is a subclass of Object
  - BUT ArrayList<Number> is not a subclass of ArrayList<Object>
- Wildcards can be lower-bounded by adding the `super` keyword
  - `public void addToOtherList(ArrayList<? super E> otherList)`
  - Accepts type E or any parent of type E

GENERIC METHODS

- Generally used for static methods
- Defines the generic type parameter at the method scope rather than the class
- Placed before the return type and after the static keyword
- Can then be used throughout the method (parameter type, return type, inner variable type)
- Examples:
  - `public static <T> void printArray(T[] array)`
  - `public static <T> T findMaxValue(T[] array)`

MORE ON GENERIC METHODS

- If I have:
  - `public static <T> T doStuff(T param1, int param2)`
- Can I do this?
  - `String result = doStuff("helloWorld", 7);`
  - Yes!
- How about this?
  - `String result = doStuff(6, 7);`
  - No! The type for T is determined by the parameter

GENERICS PRACTICE #1

- Make some changes to your merge sort method so it can work with any type

GENERICS PRACTICE #2

- Write a method that will determine the index of a provided value
- The method will take in a ADT and the value trying to be located
- Keep in mind:
  - What ADT makes the most sense
  - How you will check equivalency
  - What you will return if the value isn't found

GENERICS PRACTICE #3

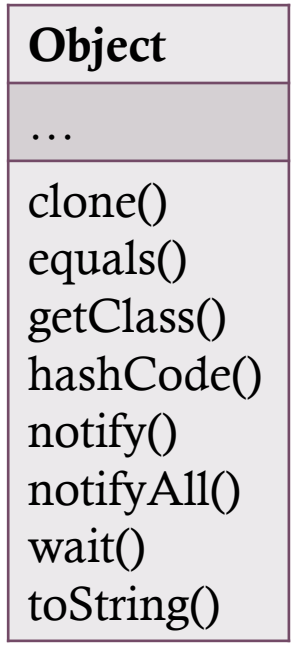
- Write the class header for a ComplexNumber class
- The class should have both a real and imaginary portion that can be any Number
- Additionally, ComplexNumber objects should be able to be sorted
  - HINT: Remember what many sort methods rely on
  - HINT: I should be able to compare a ComplexNumber object to itself

THREADS

- An execution thread (aka a lightweight process) is a sequence of instructions being executed
- Threads share memory space but have their own stack frames
- A process can have multiple threads
- So far, we've only seen one or two execution threads running at a time

THREADS IN JAVA

- Concurrency and threads are a core part of the java language (no import required!)



RUN() VS START()

- Any code we want executed by a separate thread should go in the run() method
- When it's time to generate the threads, call start()
- start() has code behind the scenes to create a new thread and execute the run() method

- Tips:
  - Do not override start(), only override run()
  - Do not call run(), unless you want everything to run on the same thread

THREAD STATES

- **NEW:** No execution thread (start() hasn't been called yet)
- **RUNNABLE:** Execution thread is running or waiting for the OS to run it
- **BLOCKED:** Waiting for a monitor lock
- **WAITING:** Waiting for another thread to wake this thread
- **TIMED_WAITING:** Waiting for a fixed amount of time to wake
- **TERMINATED:** Execution thread is finished

RUNNABLE INTERFACE

- Requires implementation of one method:
  - `public void run()`
- Allows any class (even a subclass) to have its own execution thread
- Bypasses issues with multiple inheritance

RUNNABLE INTERFACE CONTINUED

- Thread class has a constructor that takes in a Runnable object:
  - `Thread(Runnable target)`
- Assume the Clock class is defined as follows:

```
public class Clock extends JPanel implements Runnable
```

- Then, a Clock object could be passed into the Thread constructor:

```
Clock myClock = new Clock();
Thread t1 = new Thread(myClock);
t1.start();
```

- When start is called, Thread will execute myClock's run() method

UTILIZING RUNNABLE VIA LAMBDA EXPRESSION

- Can be set up using a lambda expression:

```
Runnable r2 = () -> {
    System.out.println("Runnable with Lambda Expression");
};
new Thread(r2).start();
```

THREAD PRACTICE #1

- Write a method that creates a certain number of threads, starts them, and joins them
- The method should take in a number of threads and an object to create the thread from
- The method should create Thread objects using the object parameter