Stuff you wanted to cover from Friday/Monday

- Inheritance
- Casting, dynamic method selection
  - Requested quartrice
  - Compiler vs Runtime error
- Linked lists
  - Pointers
- Recursive problems
  - Recursive Pointers
- Iterators and iterables
- How would you approach a practice problem?
  - Pattern-matching
- Comparables and comparators (from Zoom)
- Generics (from Zoom)
- 2-D arrays

Inheritance and Subclasses

General rule: A subclass is a specific "type" of the parent class.

- Ex. A "Cat" is a specific type of "Animal", so Cat can extend Animal

Because all Cats are Animals, all Cats need to be able to do everything an Animal can do.

- By default, Cat "inherits" all its methods from Animal, so Cats can do what Animals can do
  - Ex. All Animals can eat. Cat can inherit the "eat" method from Animal
- But you can choose to override the default behavior from Animal, and do Cat-specific behavior
  - If you want your cat to "eat" in a way that's different from the standard Animal behavior, you can override the "eat" method

Casting and Dynamic Method Selection (DMS)

When we press "play" on a Java program, two separate steps happen:

Compilation (all errors here are compiler errors):

- Checks for syntax errors
- Puts together a list of methods/attributes of every class
  - If a class extends another class, inherit all the methods that aren't overridden
  - If a class implements an interface, inherit all the default methods, and check to make sure that all abstract methods are overridden
  - To override, a function must have the same name AND the same arguments, which collectively forms the function *signature*. Otherwise, the override doesn't work, and both methods are kept separately.
  - Assigns a static type to every variable
  - Usually the type that the variable was declared as or the type that a method is defined to return
  - But if a typecast occurs (and the typecast is theoretically possible), use the casted type

Casting and Dynamic Method Selection (DMS)

When we press "play" on a Java program, two separate steps happen:

Compilation (all errors here are compiler errors):

- Checks for syntax errors
- Puts together a list of methods/attributes of every class
- Assigns a static type to every variable
- Checks each line *in isolation* to see if there's a way to run that line
  - Assumes the static type of every variable
- After all checks, converts the code to a computer-friendly language so it can actually be run.
- Note: Only happens once normally. After compiling once, you can run the same code repeatedly without going through the checks above, so it's faster

Casting and Dynamic Method Selection (DMS)

When we press "play" on a Java program, two separate steps happen:

Runtime (all errors here are Runtime Errors)

- Actually runs the program
- Relies on the dynamic type of the object (what the object was created as)
  - On a function call, uses the method of the same name in the dynamic type's class
    - If multiple methods have the same name (overloaded), pick the one whose signature matches
  - Finds any illegal actions that can't be found by static type analysis
  - If we lied in compilation about a typecast (ex. (Poodle) new Dog()), error
  - Array out of bounds errors, out of memory errors, etc.

Note: The hardest parts of DMS are considered out of scope for this class. In particular, we won't override an overloaded method, and we won't have "ambiguous" overloads where two methods could be applied (e.g. foo(List) and foo(ArrayList)). The above rules don't apply if we don't make these assumptions.

Example (From Spring 2024 MT 1)

```java
public interface Sphere {
     default void getRadius() { System.out.println(0); }
}

class Planet implements Sphere {
    private int radius = 5;

    @Override
    public void getRadius() { System.out.println(this.radius); }

    public void orbit(Planet p) { System.out.println("orbit planet"); }
}

class Exoplanet extends Planet {
    static int distance = 10;

    public void getDistance() { System.out.println(distance); }

    @Override
    public void orbit(Planet p) { System.out.println("orbit exoplanet"); }
}
```

```java
Class Main {
    public void main(String[] args){
        Planet kepler = new Exoplanet();

        Sphere jupiter = new Sphere();

        Sphere arion = new Exoplanet();

        Planet earth = new Planet();


        kepler.orbit((Exoplanet) arion);

        ((Exoplanet) earth).orbit(kepler);

        ((Exoplanet) kepler).getDistance();

        earth.getRadius();

        arion.getDistance();

        earth.radius;

        Exoplanet.getDistance();

    }
}
```

2D Arrays, Recursion, and Solving a Practice Problem (From Fa23 MT1)

5 I Signed an NDA

(20 Points)

High-dimensional nested arrays are quite cumbersome in Java. For example, a 9-dimensional `int` array `arr` must be declared as `int[][][][][][][][][] arr`. Angel wants to devise a class to store his $n - D$ data.

(a) Complete the constructor for `NDArray`, which takes in a dimension D and a width W, such that the `NDArray` represents a $\underbrace{W \times W \times ... \times W}_{\text{D times}}$ array. You may assume that $D >= 1$ and $W >= 1$.

```
public class NDArray {
    public int value;
    public int dimension;
    public NDArray[] arr;

    public NDArray(int D, int W) {
        dimension = D;

        if (_____1_____) { return; }

        arr = _____2_____;

        for (_____3_____) {

            _____4_____;
        }
    }
}
```

2D Arrays, Recursion, and Solving a Practice Problem

(b) Angel now needs a way to get items from the `NDArray`. Complete `get`, which is an instance method and returns the item at `List<Integer> coords`. For example, if we have a `NDArray nda` of dimension 2 representing $\begin{bmatrix}[5, & 4], \\ [1, & 9]\end{bmatrix}$ then `nda.get(List.of(0, 1))` should return `4`. You may assume that each individual coordinate is between `0` and `W-1`, inclusive. Hint: You may use the `subList` method of `List`.

```
public int get(List<Integer> coords) {
    if (coords.size() != dimension) {
        throw new IllegalArgumentException();
    }
    if (_____1_____) {

        _____2_____;
    }
    int index = coords.get(0);

    return _____3_____;
}
```

A List of Lists

Two main types:

Linked List:

- "Naked" Linked List
  - Consists of a value, and a pointer to the next element
  - Deprecated; almost never used in Java
- Singly Linked List
  - Stores a naked Linked List in an internal Node class, then "clothes" the List in the List interface
  - Only lets you move forward in the list, so less useful than DLL
  - Adds a sentinel so you don't need to worry about "empty list" edge case
- Doubly Linked List
  - Same as SLL, except each element stores a previous as well

A List of Lists

Two main types:

Linked List:

- Singly Linked List
- Doubly Linked List

Array List:

- Default array
  - Natively supported by Java (not a class)
  - Need to specify the length before using, and can't change that length
  - Not a List (because it can't grow indefinitely)
- ArrayList
  - List type that uses an array in the backend
  - Requires resizing periodically in order to account for the List interface
  - On average, constant time access (instead of linear time access in Linked List)

Example (From Spring 23 MT1)

Q5 getEveryNth

(700 Points)

In Python, we can conveniently get an array of every Nth element using the "slicing" syntax `arr[::n]`, where `n` is the "step size" of the number of elements we iterate over. For example, if `arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`, `arr[::2] = [0, 2, 4, 6, 8]`, `arr[::4] = [0, 4, 8]`, `arr[::11] = [0]`. In this question, `n` will always be a positive integer greater than 0.

We want to try to replicate this behavior in Java with our LinkedList and ArrayList classes. Fill out the code below for the two different implementations of `getEveryNth`, which is defined as follows:

- Input: A positive integer n.
- Output: A new ArrayList whose elements are the `0`th, `n`th, `2n`th, ... elements of the input List. You may NOT assume that the size of the input List is a multiple of `n`.
- Your code may NOT modify the input List.

For the purposes of this question, you may NOT use the get method of any List. You may, however, use any other functions defined as part of the List interface in the reference sheet.

Hint: the modulo operator might be useful for this question. The modulo operator returns the remainder when dividing. Some examples are attached below:

```
15 % 8 = 7 // because 15 / 8 = 1 with remainder 7
0 % 9 = 0
4 % 4 = 0
```

Example

Part A:

```
1  public class ArrayList<T> implements List<T> {
2      private T[] array;
3      private int size;
4
5      ...//More code that implements an Array List
6
7      // As a reminder, you may not use the .get method in this question
8      public List<T> getEveryNth(int n) {
9          List<T> result = new ArrayList<>();
10
11         for(____________________;____________________;____________________) {
12
13             ____________________________________________________________;
14         }
15         return result;
16     }
17 }
```

Example

```
// As a reminder, you may not use the .get method in this question
public List<T> getEveryNth(int n) {
    List<T> result = new ArrayList<>();

    Node node = ____________________________________________________;
    int counter = 0;

    while (____________________________________________________) {
        if (counter == 0) {

            ____________________________________________________;
        }

        counter = ____________________________________________________;

        ____________________________________________________;
    }
    return result;
}
```

Comparators

Designed to compare two values: either a < b, a = b, or a > b.

- For ease of use, comparisons return an int, negative if a < b, 0 if a = b, and positive if a > b

Two comparison-based interfaces:

Comparable<T> if you want to compare to a particular type

- Also useful to make a "default" comparison method.
  - ex. `public class Dog implements Comparable<Dog>` creates a Dog class that can compare itself against other dogs

Comparator<T> if you want a machine that can compare two items of a given type

- Often used when there's no "canonical ordering" of the items
  - ex. You can order Dogs by size, by name, or by any number of things, so you make a new Comparator machine for each ordering you want.
  - Often declared as inner classes

Comparators

In order for comparators to be considered valid, it must follow the following well-ordering principles:

- If a < b, then b > a. If a = b, then b = a (symmetry)
- a = a (reflexivity)
- If a < b and b < c, then a < c (transitivity)

Don't worry too much about this for now, but it'll be relevant once we get into sorts (if these aren't true, there's not necessarily a valid way to sort a set of values)

Iterators

Designed to iterate over a collection of values

- Used in the enhanced for loop (for(int i: intlist))

Two iteration-based interfaces:

Iterable<T>

- Tells the compiler that this class can be iterated over (and used in enhanced for loops)
- Requires only one function: public Iterator<T> iterator(), which returns an iterator on the elements of the class.

Iterator<T>

- The actual machine that iterates over objects
- Requires two functions:
  - public boolean hasNext(), which returns if there's more to return
  - public T next(), which returns the next item
- Similar to generators in Python, and can also be used in the same way (without a corresponding iterable object)

Generics

One distinctive feature of Java is its typing system

- All objects require a specific type on compilation, and the compiler should be able to tell what type every object is.
- This includes types that are "components" of other types.
  - Ex. We can't just create a List in Java without also saying what type of thing is stored in that list
- Java's solution is to have "generic types", which effectively act as placeholders for a type to be decided later.
  - Analogy: A higher order function is a function that takes in as input another function (like map), and acts based on that function input
  - A List is in a "higher-order class", because it takes in as input another class, and acts based on that class input
- Example: List<Integer> is a list of integers. List<String> is a list of Strings. Both can be constructed as ArrayLists, which have a class definition of ArrayList<T> (The T gets replaced by Integer and String in the above two)