Stuff you want to cover on Friday/Monday, February 12

- Static/Dynamic Type and Typecasting
- Nested Classes
- Linked Lists
- Private/Public
- Inheritance

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

https://drive.google.com/file/d/1t65JX1RtaEoSyU05hi0b9i7aPMmqLrDV/view

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 (+ How to approach Skeleton Code)

https://drive.google.com/file/d/14-IEeTJsYHvK6Ogyro4xcDT8KH_KbB9A/view

Public vs Private

General Java philosophy: The more things you're allowed to do, the more complex your code will become. To reduce code complexity and add flexibility, **consciously restrict what you're allowed to do as much as possible.**

Ex. The List interface describes a set of things that all lists can do, but you still need a class (ex. ArrayList) to actually make a list. You *can* do

```
ArrayList<Integer> a = new ArrayList<>();
```

But if you're only going to use List interface operations, it's better to do

```
List<Integer> a = new ArrayList<>();
```

This way, you can safely change to another List type just by changing one line, and you force yourself to never use ArrayList-specific operations.

Public vs Private

Access modifiers prevent other classes from "seeing" certain methods/attributes

- public: Every outside class can see and change a public variable
  - Once you build on top of a class, other code *may* rely on it, so **anything public cannot change any more, even if you want to rewrite your implementation.**
- private: Nothing outside the current class can see the variable
  - Since nothing outside relies on this, **you can safely change a private method/attribute to optimize your code.**
- Two other access levels: protected (allows subclasses and classes defined in the same folder to use the variable), and package-private (allows classes defined in the same folder to use the variable; this is the access you get if you don't specify any modifier)

Public vs Private

Access modifiers prevent other classes from "seeing" certain methods/attributes

- Nested classes are slightly different: An inner class can ALWAYS access something in the outer class, and vice versa, regardless of access modifier.
  - So access modifiers only affect how classes in *other files* see your class
- Ideal code **restricts access as much as possible**. If you can make a method or attribute private, make it private.