CS 134: Java and OOP Review

Announcements & Logistics

- Lab 10 Selection Sort in Java: due Wed/Thur @ 11 pm
- Final exam reminder: May 22 @ 9:30 am (or Wed May 18, submit form today!)
- Final exam will be cumulative: everything is fair game (including Java)
 - More weight on post-midterm topics
 - Will discuss more about this in Friday's wrap up lecture
 - Practice problems for final available on Glow (note: we didn't write these questions, but they should help you study)
- Review session/office hours next week: **check calendar!**
 - Review session: Tue, May 17, 8-9:30, TPL 203
- Course evals on Friday: bring a laptop to class if possible

LastTime

- Discussed **loops** and **conditionals** in Java
- Python for loops are most similar to for each loops in Java
- A simple Java **for loop** explicitly requires starting condition, stopping condition, and steps in the header:

```
for i in range(10):
    print(i)
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
        }
    for el in seq:
        print(el)
        ···
    }
    for (int i : myArray) {
        System.out.println(i);
        ···
    }
    for each loop in Java</pre>
```

Python vs Java: Check-in after Lab 10

• Curly braces, semicolons: what value do they add?

• Specifying **data types** at all times: how is it useful?

Python vs Java: Check-in after Lab 10

- Curly braces, semicolons: what value do they add?
 - Make the code more **maintainable** and **platform independent**!
 - White spaces, tabs, and line breaks are not stored consistently across computer architectures and operating systems
 - Converting a file from one system to another (say Windows to Mac) can change the white space
 - This would break a Python script; Java program might become unreadable but will still run!
- Specifying **data types** at all times: how is it useful?
 - In larger coding projects, not knowing the type of variables can make code harder to follow
 - This is why Python docstrings are so important!

Iodav

- Review classes, objects, and methods
 - A **class** vs an **instance** of the class (or an object of the class)
 - Attributes (or instance variables in Java) and slots
 - Accessor and mutator methods: getters and setters
 - **Scope**: public, private and protected (or _ and __ in Python)
- Note that the aforementioned topics are **language independent**!
 - We will look at them in both languages but the focus will be on reviewing the concepts and not the syntax!

Programming Language Features

- Basic features:
 - Data Types
 - Reading user input
 - Loops
 - Conditionals
- Advanced topics:



- Interfaces
- Collections
- Graphical User Interface Programming

Classes and Objects

- Classes are blueprints for **objects**
 - Collections of data (variables and attributes) and methods that act on those data
 - An **instance** is a specific realization of a class
- We did not talk about Python **classes** until Lecture 21
 - Easy to ignore/forego this topic for simple examples in Python
- In Java, all code is defined within a class
 - We have to come to terms with **classes** and **methods** from Day I
 - No such thing as a classless **module** or **function** in Java
- Support for classes are a feature of all **OOP languages**
 - Python and Java are both OOP languages

Classes and Objects

- In Python, everything is an **object**: including ints, strings, functions, etc
 - Python types are *implicit*, can be queried using type
- In Java, there are primitive types which are not objects (ints, doubles, booleans, chars etc) and "Object" versions of these types (Integer, Double, String, etc.)
 - Java requires **explicit** type declaration
- Why would we ever want to define our own classes?
 - Create our own "data types"
 - A way to bundle (or **encapsulate**) related data and methods for interacting with that data in an application-specific manner

Review: Object-Oriented Programming

Four major principles of OOP programming:

• Abstraction

• The main purpose of abstraction is hiding the unnecessary details from the users

Inheritance

• The ability for one object to take on the states, behaviors, and functionality of another object

• Encapsulation

• The bundling of data, along with the methods that operate on that data, into a single unit

Polymorphism

• Using a single type entity (method, operator or object) to represent different types in different scenarios (e.g., operator/method overloading)

Methods vs Functions

Methods (Python and Java)

- Always defined within a class
- Are called using dot notation on a specific instance of the containing class
- A method is implicitly passed a reference to the object on which it is invoked (self in Python, this in Java)
- A method can optionally manipulate parameters
- A method may or may not **return** a value
- A method can operate on the attributes/instance variables that are defined within the containing class

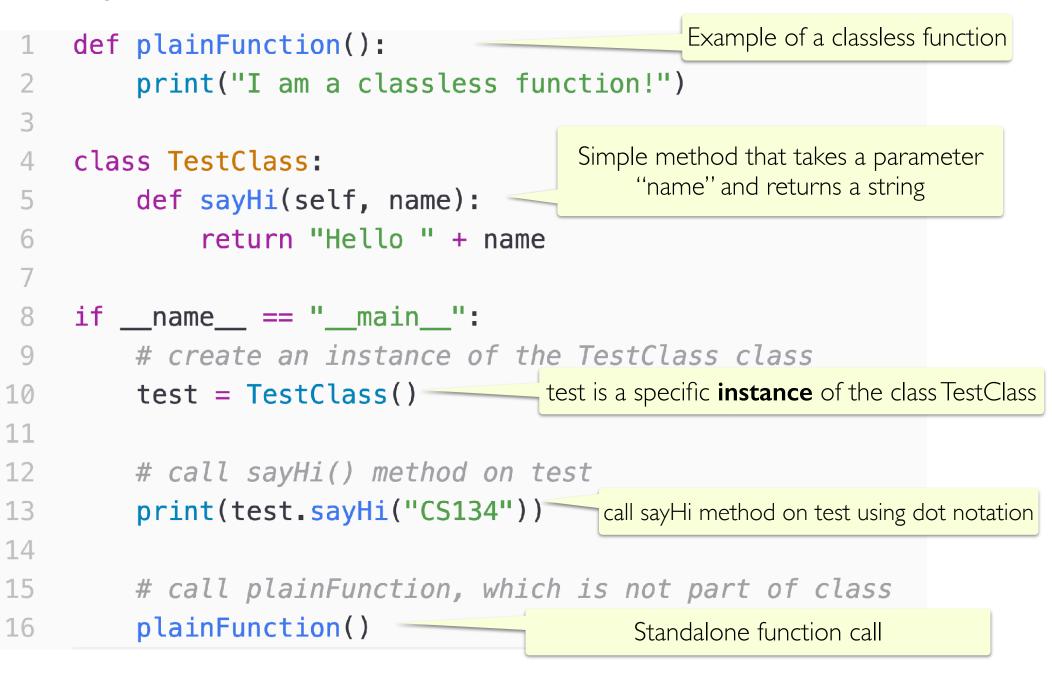
Functions (Python only)

- Stand-alone logical blocks of code that are defined outside of a class
- Once defined, a function can be called from anywhere in the program (by importing if in a separate module)
- A function definition specifies
 parameters (input that is passed to the function when it is called). If parameters are passed, they need to be passed
 explicitly
- A function may perform an action (e.g. print or modify), and/or return a value (or implicitly return None)

self Parameter Review

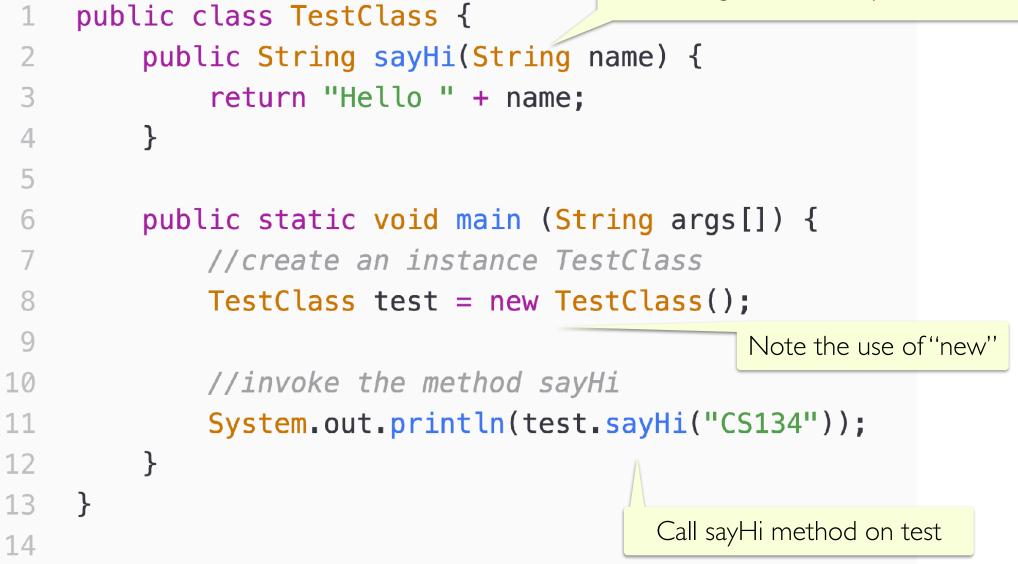
- In **Python**, method **definitions** have **self explicitly** defined as the first parameter (and we use this variable inside the method body)
- But **we don't pass the self parameter explicitly** when we *invoke* the methods!
- This is because whenever we call a method on an object, the object itself is **implicitly** passed as the first parameter
- Methods are like **object-specific functions** and this lets us access the object's attributes via the methods directly

Python Class, Methods, & Function



Java Class and Methods

Method that returns a String and takes a String "name" as a parameter



Data Attributes or Instance Variables

- Classes keep track of relevant state in instance variables (Java) or attributes (Python)
- In Python, attributes should be stored in ____slots____
 - Attributes in **____slots**___ (list of strings) are explicitly specified
- In Java, **instance variables** are typically defined at the top of the class before all methods
 - Instance variables are accessible to all methods of the class
- RULE OF THUMB: Make all attributes private (or protected)
 - In Python, this means using "_" or "__" and in Java we say ''**private**''
 - Only accessed via accessor (**getter**) and mutator (**setter**) methods

Scope Review

Private

- Python: Double leading underscore
 (___) in name of variable or method
- Java: Use the keyword private
- Private methods and variables/attributes are not accessible from outside of the containing class

Protected

- Python: Single leading underscore (_) in name of variable or method
- Java: Use the keyword protected
- Protected methods and variables/ attributes should only be accessed by subclasses

Public

- **Python:** No leading underscore in name of variable or method
- Java: Use the keyword public
- Public methods and variables/attributes
 can be *freely used outside of the class*

These access rules are actually enforced in Java; are more of a convention in Python

Methods and Data Abstraction

- Users are given access to data attributes only through methods in OOP
- Manipulating attributes/instance variables should only be done via:
 - **accessor (getter) methods:** provide 'read-only' access to the class attributes/instance variables (return value)
 - **mutator (setter) methods:** let us modify the values of class attributes/instance variables (do not return)
- Using getters and setters enforces **data abstraction**
 - Methods provide a **public interface** to attribute values
 - Attribute representation remains part of the *private implementation*

-	<pre># originally in lec 27 class LinkedList: """Implements our own recursive list data structure""" slots = ['_value', '_rest'] Private attributes</pre>		
	<pre>definit(self, value=None, rest=None): selfvalue = value selfrest = rest</pre>		
)	# getters/setters		
-	<pre>def getRest(self):</pre>		
)			
	<pre>def getValue(self): return selfvalue </pre>		
)	<pre>def setValue(self, val): selfvalue = val public setter method for _value</pre>		

```
public class LinkedList {
 1
                                                     Private instance variables
       private String value;
 2
                                           Notice that rest is of type LinkedList. Recursion!
 3
       private LinkedList rest;
 4
       public LinkedList(String val) {
 5
         this.value = val:
 6
         this.rest = null;
                                                           Constructors, like __init__ in
 7
                                                              Python. Ignore for now!
       }
 8
 9
       public LinkedList(String val, LinkedList other) {
10
         this.value = val:
11
         this.rest = other;
12
13
       }
14
15
       public String getValue() {
                                                       public getter method for value
         return this.value;
16
       }
17
18
19
       public LinkedList getRest() {
                                                        public getter method for rest
         return this.rest;
20
       }
21
22
       public void setValue(String v) {
23
                                                       public setter method for value
         this.value = v:
24
25
       }
```

Special Methods & Operator Overloading

- Classes in Python and Java define several "special" methods
 - Python: __init__, __str__, __eq__
 - Java: constructor(s), toString(), equals()
- Python has many more due to **operator overloading**
 - Operator overloading means we redefine common operations (like addition + or using list notation [] for access) for our data type
 - __add__, __getitem__, __setitem__, __contains__
 - Many more!
- Java does not support operator overloading
 - But it does support **method overloading** (same method, different parameters)

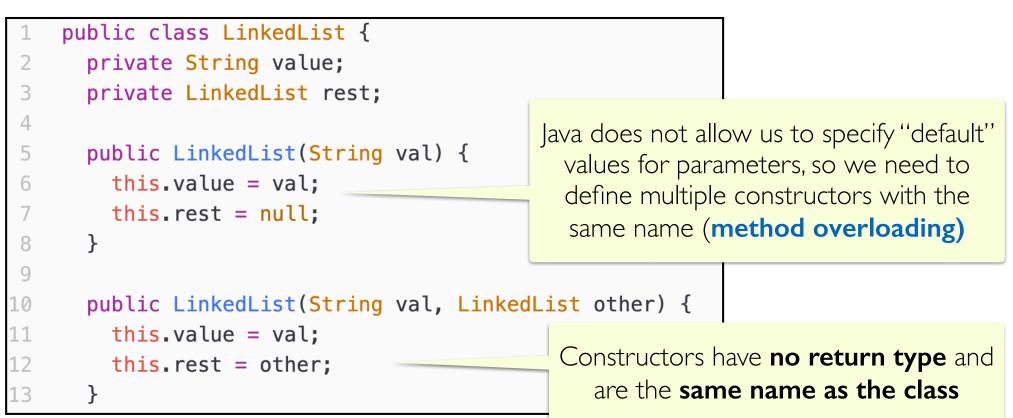
Initializing an Object

- When creating a new instance of a class in Python or Java, we have to initialize the values of the attributes/instance variables
 - Python: __init__ method
 - Java: Constructor(s)
- These special methods are **automatically called** when you create an instance of the class
 - Python: board = BoggleBoard()
 - Java: BoggleBoard board = new BoggleBoard() (notice the use of new)
- Let's look at how this works for our LinkedList

Python

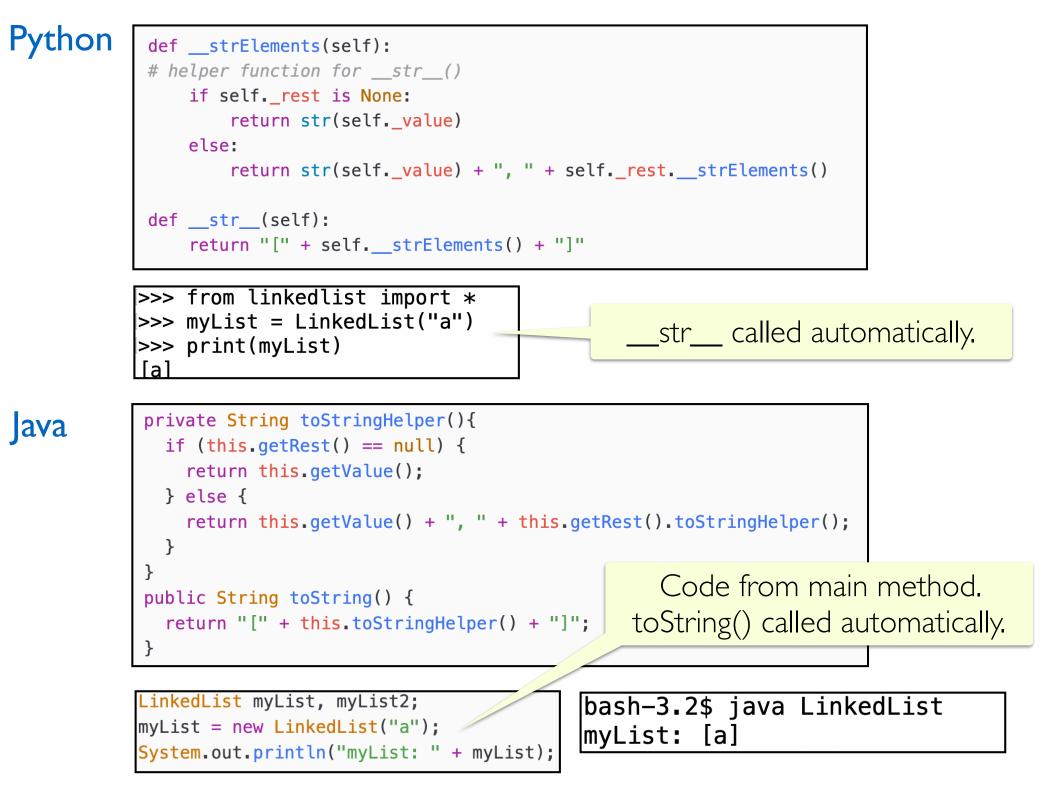
```
2 class LinkedList:
3 """Implements our own recursive list data structure"""
4 __slots__ = ['_value', '_rest']
5 
6 def __init__(self, value=None, rest=None):
7 self._value = value
8 self._rest = rest
```

Java



String Representation of an Object

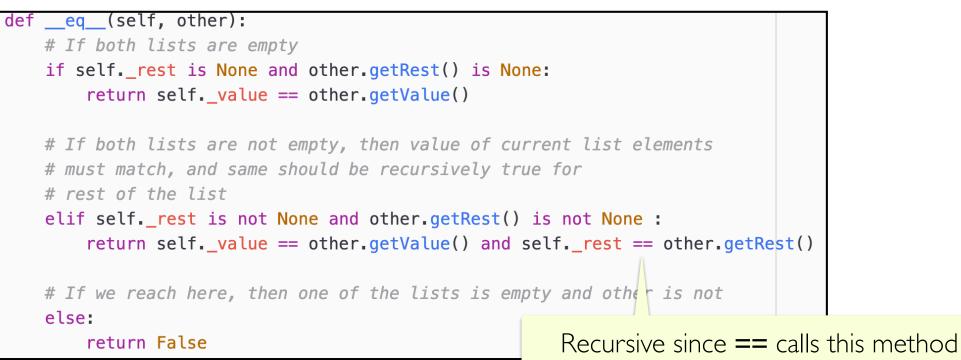
- It is often convenient to be able to print a string "version" of an instance of a class
 - Very helpful when debugging
- Python and Java both provide special methods for this
 - Python: ____str___ and ___repr___
 - Java: toString()
- For <u>str</u> and toString(), we can choose how the objects of the class are printed
- For <u>repr</u> (Python only), we want to generate a string that would allow us to recreate the object



Comparing Objects

- Often convenient to compare two instances of a class
- We have to decide if we want to compare their **values** or **identities**
- Comparing **values**: determining if the data contained in two separate instances of a class is the same (e.g., two lists that contains same values)
 - Python: == operator (___eq___ special method, operator overloading)
 - Java: equals() method
- Comparing identities: determining if two instances are actually the same? (Do they reside in the same place in memory?)
 - **Python: is** operator (**cannot be overloaded!**)
 - Java: == operator

Python



ava

<pre>return true; } else if (this.getRest() boolean val = this.get</pre>	<pre>edList other) { ll && other.getRest() == null) { != null && other.getRest() != null) { etValue().equals(other.getValue()); Rest().equals(other.getRest()); </pre>	Generally speaking in Java, we use equals() to compare anything other than primitive types. Be careful using == with objects in Java!
return false; }	Recursive call to equals()	

Other Useful Methods

- **Testing membership** we often want to know if a specific item or value exists in our data structure
 - **Python: in** operator (<u>contains</u> special method)
 - Java: contains() method
- **Computing length** we often want to know the length or size of a data structure
 - **Python: len** function (<u>len</u> special method)
 - Java: length() method
- For our LinkedList implementations, all of these operations/ methods will be recursive

Other Useful Methods

Python

Java

```
public int length() {
# len() function calls len () method
                                                            if (this.getRest() == null && this.getValue() == null) {
# slightly updated version that accounts for empty list
def len (self):
                                                              return 0;
   # base case: i'm an empty list
                                                            } else if (this.getRest() == null) {
    if self._rest is None and self._value is None:
                                                               return 1;
        return 0
                                                            } else {
    # i am the last item
                                                              return 1 + this.getRest().length();
   elif self._rest is None and self._value is not None:
                                                            }
        return 1
                                                          }
    else:
       # same as return 1 + self._rest.__len__()
                                                          public boolean contains(String search) {
       return 1 + len(self._rest)
                                                            if (this.getValue().equals(search)){
                                                              return true;
# in operator calls contains () method
                                                            } else if (this.getRest() == null) {
def contains (self, val):
                                                              return false;
    if self._value == val:
                                                            } else {
        return True
                                                              return this.getRest().contains(search);
    elif self. rest is None:
                                                            }
        return False
    else:
       # same as calling self. contains (val)
        return val in self._rest
```