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

Last Time

- 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 el in seq:
    print(el)
    ...
```

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);

    ...
}

for (int i : myArray) {
    System.out.println(i);

    ...
}
```

for each loop in Java

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!

Today

- 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:**
  - Classes
  - 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 1
  - 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

```
1  def plainFunction():
2      print("I am a classless function!")
3
4  class TestClass:
5      def sayHi(self, name):
6          return "Hello " + name
7
8  if __name__ == "__main__":
9      # create an instance of the TestClass class
10     test = TestClass()
11
12     # call sayHi() method on test
13     print(test.sayHi("CS134"))
14
15     # call plainFunction, which is not part of class
16     plainFunction()
```

Example of a classless function

Simple method that takes a parameter "name" and returns a string

test is a specific **instance** of the class TestClass

call sayHi method on test using dot notation

Standalone function call

Java Class and Methods

```
1  public class TestClass {
2      public String sayHi(String name) {
3          return "Hello " + name;
4      }
5
6      public static void main (String args[]) {
7          //create an instance TestClass
8          TestClass test = new TestClass();
9
10         //invoke the method sayHi
11         System.out.println(test.sayHi("CS134"));
12     }
13 }
14
```

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

Note the use of "new"

Call sayHi method on test

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***

```
1  # originally in lec 27
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
9
10     # getters/setters
11     def getRest(self):
12         return self._rest
13
14     def getValue(self):
15         return self._value
16
17     def setValue(self, val):
18         self._value = val
19
```

Private attributes

public getter method for _rest

public getter method for _value

public setter method for _value

```java
public class LinkedList {
  private String value;
  private LinkedList rest;

  public LinkedList(String val) {
    this.value = val;
    this.rest = null;
  }

  public LinkedList(String val, LinkedList other) {
    this.value = val;
    this.rest = other;
  }

  public String getValue() {
    return this.value;
  }

  public LinkedList getRest() {
    return this.rest;
  }

  public void setValue(String v) {
    this.value = v;
  }
```

Private instance variables
Notice that rest is of type LinkedList. Recursion!

Constructors, like __init__ in Python. Ignore for now!

public getter method for value

public getter method for rest

public setter method for value

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

```
1  public class LinkedList {
2    private String value;
3    private LinkedList rest;
4
5    public LinkedList(String val) {
6      this.value = val;
7      this.rest = null;
8    }
9
10   public LinkedList(String val, LinkedList other) {
11     this.value = val;
12     this.rest = other;
13   }
```

Java does not allow us to specify "default" values for parameters, so we need to define multiple constructors with the same name (**method overloading)**

Constructors have **no return type** and are the **same name as the class**

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 `__str__` and `toString()`, we can choose how the objects of the class are printed
- For `__repr__` (Python only), we want to generate a string that would allow us to recreate the object

Python

```
def __strElements(self):
# helper function for __str__()
    if self._rest is None:
        return str(self._value)
    else:
        return str(self._value) + ", " + self._rest.__strElements()

def __str__(self):
    return "[" + self.__strElements() + "]"
```

```
>>> from linkedlist import *
>>> myList = LinkedList("a")
>>> print(myList)
[a]
```

__str__ called automatically.

Java

```
private String toStringHelper(){
  if (this.getRest() == null) {
    return this.getValue();
  } else {
    return this.getValue() + ", " + this.getRest().toStringHelper();
  }
}
public String toString() {
  return "[" + this.toStringHelper() + "]";
}
```

Code from main method.
toString() called automatically.

```
LinkedList myList, myList2;
myList = new LinkedList("a");
System.out.println("myList: " + myList);
```

```
bash-3.2$ java LinkedList
myList: [a]
```

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

```
def __eq__(self, other):
    # If both lists are empty
    if self._rest is None and other.getRest() is None:
        return self._value == other.getValue()

    # If both lists are not empty, then value of current list elements
    # must match, and same should be recursively true for
    # rest of the list
    elif self._rest is not None and other.getRest() is not None :
        return self._value == other.getValue() and self._rest == other.getRest()

    # If we reach here, then one of the lists is empty and other is not
    else:
        return False
```

Recursive since == calls this method

Java

```
public boolean equals(LinkedList other) {
  if (this.getRest() == null && other.getRest() == null) {
      return true;
  } else if (this.getRest() != null && other.getRest() != null) {
      boolean val = this.getValue().equals(other.getValue());
      boolean r = this.getRest().equals(other.getRest());
      return val && r;
  } else {
    return false;
  }
}
```

Recursive call to `equals()`

Generally speaking in Java, we use `equals()` to compare anything other than primitive types. Be careful using == with objects in Java!

Other Useful Methods

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

Other Useful Methods

Python

```python
# len() function calls __len__() method
# slightly updated version that accounts for empty list
def __len__(self):
    # base case: i'm an empty list
    if self._rest is None and self._value is None:
        return 0
    # i am the last item
    elif self._rest is None and self._value is not None:
        return 1
    else:
        # same as return 1 + self._rest.__len__()
        return 1 + len(self._rest)

# in operator calls __contains__() method
def __contains__(self, val):
    if self._value == val:
        return True
    elif self._rest is None:
        return False
    else:
        # same as calling self.__contains__(val)
        return val in self._rest
```

Java

```java
public int length() {
  if (this.getRest() == null && this.getValue() == null) {
    return 0;
  } else if (this.getRest() == null) {
    return 1;
  } else {
    return 1 + this.getRest().length();
  }
}

public boolean contains(String search) {
  if (this.getValue().equals(search)){
    return true;
  } else if (this.getRest() == null) {
    return false;
  } else {
    return this.getRest().contains(search);
  }
}
```