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:

```python
for i in range(10):
    print(i)

...  

for el in seq:
    print(el)

...  
```

```java
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
1 def plainFunction():
   print("I am a classless function!")

2
class TestClass:
3     def sayHi(self, name):
4         return "Hello " + name

5
6 if __name__ == "__main__":
7     # create an instance of the TestClass class
8     test = TestClass()

9     # call sayHi() method on test
10    print(test.sayHi("CS134"))

11
12    # call plainFunction, which is not part of class
13    plainFunction()
14
15```
Java Class and Methods

```java
public class TestClass {
    public String sayHi(String name) {
        return "Hello " + name;
    }
}

double the value of test = new TestClass();

// invoke the method sayHi
System.out.println(test.sayHi("CS134"));
```
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

<table>
<thead>
<tr>
<th>Private</th>
<th>Protected</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Python:</strong> Double leading underscore (__) in name of variable or method</td>
<td><strong>Python:</strong> Single leading underscore (_ ) in name of variable or method</td>
<td><strong>Python:</strong> No leading underscore in name of variable or method</td>
</tr>
<tr>
<td><strong>Java:</strong> Use the keyword <code>private</code></td>
<td><strong>Java:</strong> Use the keyword <code>protected</code></td>
<td><strong>Java:</strong> Use the keyword <code>public</code></td>
</tr>
<tr>
<td>Private methods and variables/attributes are <strong>not accessible from outside</strong> of the containing class</td>
<td>Protected methods and variables/attributes should <strong>only be accessed by subclasses</strong></td>
<td>Public methods and variables/attributes can be <strong>freely used outside of the class</strong></td>
</tr>
</tbody>
</table>

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**
```python
# originally in lec 27

class LinkedList:
    """Implements our own recursive list data structure"""
    __slots__ = ['_value', '_rest']

    def __init__(self, value=None, rest=None):
        self._value = value
        self._rest = rest

    # getters/setters
    def getRest(self):
        return self._rest

    def getValue(self):
        return self._value

    def setValue(self, val):
        self._value = val
```

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;
    }
}
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

```python
class LinkedList:
    """Implements our own recursive list data structure"""
    __slots__ = ['_value', '_rest']

    def __init__(self, value=None, rest=None):
        self._value = value
        self._rest = rest
```

Java

```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;
    }
}
```

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**

```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() + "]"
```

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

**Java**

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

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

```bash
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 contain the 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

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;
    }
}
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._rest.__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);
    }
}
```