Homework 8

This homework covers the following topics from class:

- OOP palooza, part 2
    - Classes (Encapsulation/Access Modifiers, This and Self, Properties), Inheritance Approaches part 1 (Interface Inheritance)
- OOP palooza, part 3
    - Inheritance Approaches cont. (Subclassing, Implementation, Prototypal), Inheritance Topics (Construction/Destruction/Finalization, Method Overriding, Multiple Inheritance)

Each problem below has a time associated with it. This is the amount of time that we expect a thoroughly-prepared student would take to solve this problem on an exam. We understand, however, that as you learn these concepts, these questions may take more time to solve, and we don't want to overwhelm you with homework. For that reason, only **required** questions need to be completed when you submit this homework. That said, the exams will cover all materials covered (a) in class, (b) from class projects, (c) in the required homework problems, and (d) in the optional homework problems, so at a minimum, please review the optional problems and/or use them as exam prep materials.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and hand-written solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work.

For homework #8, you must complete the following **required** problems:

OOP1: Classes, Objects (5 min)
OOP2: Classes, Objects, Encapsulation (5 min)
OOP3: Interfaces and Types (5 min)
OOP4: Classes, Getters, Setters (5 min)
OOP5: Classes, Interfaces in Dynamically-typed Languages (5 min)
OOP6: Subclass Inheritance, Interface Inheritance (5 min)

OOP7: Interface Inheritance, Supertypes and Subtypes (8 min)

For homework #8, you may optionally complete the following additional problems:

N/A

OOP1: Classes, Objects (5 min)

(5 min.) As we learned in class, some of the original OOP languages didn't support classes, just objects. As we know, classes serve as "blueprints" for creating new objects, enabling us to easily create many like objects with the same set of methods and fields. In a language without classes, like JavaScript, how would you go about creating uniform sets of objects that all have the same methods/fields?

Your answer:

OOP2: Classes, Objects, Encapsulation (5 min)

(5 min.) By convention, member variables in Python objects are typically accessed directly without using accessor or mutator methods (i.e., getters and setters) - even by code external to a class. What impact does this have on encapsulation and why might Python

have chosen this approach? Are there ever cases when you should use getters/setters in Python rather than allowing direct access to member variables? If so, when?

Your answer:

OOP3: Interfaces and Types (5 min)

(5 min.) We learned in class that when we define a new interface X, this also results in the creation of a new *type* called X. So in the following code, the definition of the abstract IShape class (C++'s equivalent of an interface) defines a new type called IShape:

```
// IShape is an abstract base class in C++, representing an
interface
class IShape {
public:
  virtual double get_area() const = 0;
  virtual double get_perimeter() const = 0;
};
```

Now consider the code below:

```
int main() {
 IShape *ptr;  // works!

 IShape x;     // does it work?
```

```
}
```

Can the *IShape* type be used to instantiate concrete objects like x or only pointers like ptr? Why?

Your answer:

OOP4: Classes, Getters, Setters (5 min)

(5 min.) In Java, the *protected* keyword has a different meaning than in languages like C++. Here's an example of two unrelated classes that are defined in the same "package" (a package is in some ways like a namespace in C++):

foo.java

```
package edu.ucla.protected_example;

class Foo {
    protected int myProtectedVariable;

    public Foo() {
        myProtectedVariable = 42;
    }
}
```

bar.java

```
package edu.ucla.protected_example;

class Bar {
    public void accessProtectedVariable(Foo f) {
        System.out.println("Accesses Foo's protected var: " +
                           f.myProtectedVariable);
    }
}
```

Inspecting the above code, we can see that the Bar class is able to access the protected members of the Foo class, even though the two classes are unrelated (except by virtue of the fact that they are part of the same package). How does this differ from the semantics of the *protected* keyword in C++, and what are the pros and cons of Java's approach to *protected* relative to C++?

Your answer:

OOP5: Classes, Interfaces in Dynamically-typed Languages (5 min)

(5 min.) Some dynamically-typed languages support interfaces, but they serve a different purpose than interfaces in statically-typed languages. Explain what purpose interfaces

serve in dynamically-typed languages and how this differs from their use in statically-typed languages?

Your answer:

OOP6: Subclass Inheritance, Interface Inheritance (5 min)

(5 min.) Give an example of where you'd prefer interface inheritance instead of traditional subclass inheritance. Why? Give an example of when you'd want to do the opposite.

Your answer:

OOP7: Interface Inheritance, Supertypes and Subtypes (8 min)

Consider the following class and interface hierarchy:

```
interface A {
   ... // details are irrelevant
}

interface B implements A {
 ... // details are irrelevant
}

interface C implements A {
 ... // details are irrelevant
}

class D implements B, C
 ... // details are irrelevant
}

class E implements C
 ... // details are irrelevant
}

class F implements D {
}

class G implements B {
}
```

Part A: (5 min.) For each interface and class, A through F, list all of the supertypes for that interface or class. (e.g., "Class E has supertypes of A and B")

Your answer:

Part B: (3 min.) Given a function:

```
void foo(B b) { … }
```

which takes in a parameter of type B, which of the above classes (D - G) could have their objects passed to function `foo()`?

Your answer:

Part C: (1 min.) Given a function:

```
void bar(C c) { … }
```

Can the following function `bletch` call `bar`? Why or why not?

```
void bletch(A a) {
    bar(a);  // does this work?
}
```

Your answer: