Homework 8 Solutions

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)

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?

Solution:

We could create a factory method which creates and returns a new object, and use this factory method to instantiate all new objects of a particular kind (e.g., Dog objects). This would ensure that all new objects will have a consistent set of fields and methods. Here's an example that ChatGPT generated for us:

```
function createDog(name, weight) {
  return {
    name: name,
    weight: weight,
    bark: function() {
      console.log("Woof woof!");
    },
    bite: function() {
      console.log("The dog bites!");
    }
  };
}

// Instantiate two Dog objects
const dog1 = createDog("Buddy", 20);
const dog2 = createDog("Max", 15);

// Check fields
console.log(dog1.name);     // Output: Buddy
console.log(dog1.weight);   // Output: 20
console.log(dog2.name);     // Output: Max
console.log(dog2.weight);   // Output: 15

// Invoke methods
dog1.bark();   // Output: Woof woof!
dog1.bite();   // Output: The dog bites!
dog2.bark();   // Output: Woof woof!
dog2.bite();   // Output: The dog bites!
```

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?

Solution (adapted from ChatGPT):

This convention has an impact on encapsulation, which is the principle of bundling data and behavior together within an object while hiding the internal details.

The convention of direct access to member variables in Python can be seen as a departure from strict encapsulation because it allows external code to directly manipulate the internal state of an object. This approach is often referred to as ”public attributes” or ”public data” in Python.

Python may have chosen this approach for a few reasons:

- Simplicity and readability: Direct access to member variables simplifies code and makes it more readable. Python emphasizes simplicity and readability as part of its design philosophy, and direct access to member variables aligns with this principle.
- Flexibility and dynamism: Python is a dynamically-typed language, which means that the type of an object can change at runtime. By allowing direct access to member variables, Python provides flexibility to modify the internal state of an object dynamically, which can be useful in certain scenarios.

While direct access to member variables is the convention in Python, there may still be cases where using getters and setters is beneficial:

- Hiding internal state: When a member variable is less of an attribute (e.g. name, age) and instead represents internal state (which could change during a refactor), code external to the class should never directly access it.
- Validation: Getters and setters can be used to enforce validation (e.g., ensuring a field is not set to an invalid value) or apply additional logic when accessing or modifying member variables. For example, you may want to validate that a certain value falls within a specific range or trigger some action when a value is set.
- Compatibility and integration: Using getters and setters can ensure compatibility and integration with existing code or external libraries that expect access to be performed through methods rather than direct attribute access.

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?

Solution:

The type associated with an interface is a ”reference type” and it can only be used to create references, pointers and object references (depending on what the language supports). Why? Because the interface doesn’t have implementation(s) for its methods, and so if you were to define an object, e.g.

IShape x;

The compiler knows that it’s an incomplete object, and that if you were to later try to call one of its methods, e.g.:

x.get_area();

That this would fail. Rather than allowing the programmer to define an incomplete object (that’s missing implementations for its methods) and detecting the error upon a later attempt to use the object, compilers opt to prevent creation of concrete objects in the first place.

In contrast, we can use types defined by interfaces for references, object references and pointers, since these only need to *point to*, or *refer to*, other objects that *do* fully support the interface.

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++?

Solution (adapted from ChatGPT):

In Java, the protected keyword is used as an access modifier to restrict access to class members (fields, methods, constructors) within the same package or within subclasses, regardless of the package they belong to. This means that a protected member can be accessed by other classes in the same package and by subclasses, (even if they are in different packages).

On the other hand, in C++, the protected keyword provides access within subclasses only. This means that a protected member can be accessed by subclasses, but not by other classes in the same file or by unrelated classes.

The difference in the semantics of the protected keyword between Java and C++ can be summarized as follows:

- Java's protected allows access within the same package and by subclasses, regardless of the package they belong to.
- C++'s protected allows access only within subclasses.

Now, let's discuss the pros and cons of Java's approach to protected relative to C++:

Pros of Java's approach:

- Flexibility: Java's "protected" allows for a more flexible and encapsulated design, as it provides access to members within subclasses *and* within the same package. So you have more options as a developer.
- Code Organization: Java's approach supports package-level encapsulation. It supports organizing related classes within the same package and allows them to access each other's protected members, providing a way to manage the visibility and access to those members within a logical unit.

Cons of Java's approach:

- Reduced Access Restriction: Java's protected provides wider access compared to C++. It allows access within the same package, which may increase the risk of unintended access or modifications by unrelated classes within the package, compromising encapsulation.
- Increased Complexity: The wider access provided by Java's protected can make it harder to reason about the visibility and access control of class members, especially in larger codebases. It may require additional attention to ensure that only the intended classes can access and modify protected members.

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?

Solution:

An interface defines a set of method prototypes that a class must implement. Every time you define a new interface, you also implicitly define a new type.

In a statically typed language, we can indicate that a parameter to a function must be of a certain type, and this indicates what interface that the parameter variable is guaranteed to support, and thus what methods we can call on the object, e.g.,

```
void process(IWashable obj) {
  obj.wash();
  obj.dry();
}
```

implies that the object passed in supports the IWashable interface, which can be used to wash() and dry() objects.

In dynamically-typed languages parameters/variable's don't have types. As such, there's no way to specify that a particular function requires values passed to it to support a particular interface associated with any particular type. Consider the same sort function in Python:

```
def process(obj):
   obj.wash()   # duck typing
   obj.dry()    # duck typing
```

Since we don't specify the type of variables, like obj, there's no way to specify that a variable must implement the methods contained in a specific interface like IWashable. Instead, in a dynamically typed language, we'd just use duck typing to invoke the wash() and dry() methods in each object.

So while we can have interfaces in some dynamically typed languages (e.g., with Python via Abstract Base Classes), they are not used to define the types of variables/parameters and thus determine the sets of operations that may be performed on those variables. In dynamically-typed languages, interfaces are simply used to specify a minimal set of required methods that a class must implement in order to be compliant with the interface.

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.

Solution (adapted from ChatGPT):

Interface inheritance and subtype inheritance serve different purposes and are applicable in different scenarios. Let's explore examples where you would prefer one over the other:

Example 1: Interface Inheritance

Suppose you are designing a system that involves various types of movable objects, such as cars and sharks. Each movable object can have different behaviors, such as how they turn, slow, and accelerate, however these objects are otherwise unrelated (e.g., sharks and cars can both move, but are unrelated). In this case, you may choose to use interface inheritance to define a common interface, such as MotionControls, that declares these behaviors:

```
interface MotionControls {
    void turn(double angle);
    void slow(double m_s_s);
    void accelerate(double m_s_s);
}
```

```
class Car implements MotionControls {
    // Implement turn(), slow(), accelerate() specific to cars
}

class Shark implements MotionControls {
    // // Implement turn(), slow(), accelerate() specific to sharks
}
```

Here, interface inheritance allows you to group different types of movable objects under a common interface, enabling code that interacts with vastly different types of objects to operate generically without needing to know the specific type.

Now here's an example where subclassing inheritance is warranted - we want our subclasses to inherit not only a public interface, but also implementations of methods/fields:

```
class Animal {
    String name;

    Animal(String name) {
        this.name = name;
    }

    public void makeSound() {
        System.out.println("The animal makes a sound.");
    }

    public String getName() {
        return name;
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println("The dog barks.");
    }

    public void fetch() {
        System.out.println("The dog fetches a ball.");
    }
}
```

Here Animals not only define a public interface (getName and makeSound) but they also have implementations that may be inherited by subclasses (e.g., the { body } of getName).

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")

Solution:

A: No supertypes

B: Has a supertype of A

C: Has a supertype of A

D: Has supertypes of A, B and C

E: Has supertypes of A and C

F: Has supertypes of A, B, C and D

G: Has supertypes of B and A

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()`?

Solution:

Any object which has a supertype of B can be passed to the foo() function. This includes: D, F and G

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?
}
```

Solution:

No. While every C object is a subtype of A, not every A object is a subtype of C. So we can't pass an A in where a C is expected.