Homework 8 – Spring 2023 (Due: May 31, 2023)

In this homework, you'll explore more concepts from OOP palooza: abstract classes, classes and types, subtype polymorphism, dynamic dispatch, and SOLID OOP design principles. Some questions have multiple, distinct answers which would be acceptable, so there might not be a "right" answer: what's important is your ability to justify your answer with clear and concise reasoning that utilizes the appropriate terminology discussed in class. Each question has a time estimate; you'll know you're ready for the exam when you can solve them roughly within their time constraints.

We understand, however, that as you learn these questions may take more time. For that reason, only **starred red** questions need to be completed when you submit this homework (the rest should be used as exam prep materials). Note that for some multi-part questions, not all parts are marked as red so you may skip unstarred subparts in this homework.

You must turn in a PDF file with your answers via Gradescope - you may include both typed and handwritten 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.

1. ** Consider the following class and interface hierarchy:

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

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

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

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

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

class F extends D {
}

class G implements B {
}
```

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

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

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.

3. ** (5 min.) Explain the differences between inheritance, subtype polymorphism, and dynamic dispatch.

Solution:

Inheritance is where one class inherits interfaces, implementations or both from a base class.

Subtype (Subclass) Polymorphism is where I pass a subtype object to a function that expects a supertype (e.g., passing a Dog to a function that accepts Mammals). It's about the typing relationship that allows the subtype object to be substituted for a supertype object. SP is only used in statically-typed languages, since variables don't have types in dynamically typed languages.

Dynamic dispatch is where a program determines which function to call at runtime, by inspecting the object's type and figuring out the proper function to call. Dynamic dispatch is used in both statically typed languages (for virtual functions) and dynamically typed languages (for duck typing).

4. ** (5 min.) Explain why we don't/can't use subtype polymorphism in a dynamically-typed language. Following from that, explain whether or not we can use dynamic dispatch in a dynamically-typed language. If we can, give an example of where it would be used. If we can't, explain why.

Solution: Subtype polymorphism requires us to use a *super-type variable* to refer to *a subtype object*, as we see here:

```
public void foo(Shape s) {
  System.out.println(s.area());
}
public void bar() {
  Circle c = new Circle(10);
  foo(c);        // uses subtype polymorphism

  Shape s  = c; // also uses subtype polymorphism
  System.out.println(s.area());
}
```

Since in dynamically-typed languages variables don't have types, (only values have types) we cannot possibly refer to a subtype object via a supertype variable.

However, we can and must use dynamic dispatch in dynamically-typed languages.

Any time we call a method on an object, the language uses dynamic dispatch to determine the proper method to call (using a vtable embedded in the object).

5. (5 min.) Consider the following classes:

```
class SuperCharger {
public:
  void get_power() { ... }
  double get_max_amps() const { ... }
  double check_price_per_kwh() const { ... }
};

class ElectricVehicle {
public:
  void charge(SuperCharger& sc) { ... }
};
```

Which SOLID principle(s) do these classes violate? What would you add or change to improve this design?

Solution: The ElectricVehicle class violates the Dependency Inversion Principle. Rather than having an ElectricVehicle's charge() method directly take a SuperCharger object as its parameter, we should define an interface, e.g.:

```
class ICharger {
public:
  virtual void get_power() = 0;
  virtual double get_max_amps() const = 0;
  virtual double check_price_per_kwh() const = 0;
};
```

Then define our SuperCharger using this interface:

```
class SuperCharger: public ICharger { ... }
```

And finally update our ElectricVehicle class to take a Charger interface rather than a SuperCharger class.

```
class ElectricVehicle {
public:
  void charge(ICharger& sc) { ... }
};
```

This way we could define other chargers, e.g. a CheapCharger and use it to charge our ElectricVehicle too:

```
class CheapCharger: public ICharger { … }
```

The ElectricVehicle class could also be said to violate the open/closed principle since our ElectricVehicle class is tied to a specific charger - the SuperCharger, and can't work with other chargers should we define them. Migrating to an interface, as we did above, would help solve this problem. Or we could define a Charging base class, and then derive our SuperCharge and CheapCharger from that. Our ElectricVehicle's charge() method would then accept a reference to that Charging base class rather than a specific charger like a SuperCharger.

6. (10 min.) Does the Liskov substitution principle apply to dynamically-typed languages like Python or JavaScript (which doesn't even have classes, just objects)? Why or why not?

   Solution: Yes - it does apply. The LSP states that an object of a particular class may be replaced by an object of a subclass without breaking the code that uses those objects. The definition does not require that we have subtype polymorphism. We have superclasses and subclasses in many dynamically-typed languages, and if we pass a subclass object to a function that otherwise works on supertype objects, the function should work as expected.