Homework 7 – Fall 2022 (Due: Dec 1, 2022)

In this homework, you'll explore more concepts from OOP palooza and Control palooza: abstract classes, classes and types, subtype polymorphism, dynamic dispatch, and SOLID OOP design principles, short circuiting, and looping/iteration and different types of iterators.. 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. ** (10 min.) Consider the following if-statements which evaluates both AND and OR clauses:

```
if (e() || f() && g() || h())
  do_something();
if (e() && f() || g() && h())
  do_something();
if (e() && (f() || g())
  do_something();
```

How do you think short-circuiting will work with such an expression with both boolean operators and/or parenthesis? Try out some examples in C++ or Python to build some intuition. Give pseudocode or a written explanation.

2. ** 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”)

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()`?

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?
}
```

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

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.

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?

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?

7. ** For this problem, you will be using the following simple Hash Table class and accompanying Node class written in Python:

```
class Node:
 def __init__(self, val):
   self.value = val
   self.next = None

class HashTable:
 def __init__(self, buckets):
   self.array = [None] * buckets

 def insert(self, val):
   bucket = hash(val) % len(self.array)
   tmp_head = Node(val)
   tmp_head.next = self.array[bucket]
   self.array[bucket] = tmp_head
```

a) ** (10 min.) Write a Python generator function capable of iterating over all of the items in your HashTable container, and update the HashTable class so it is an iterable class that uses your generator.

b) ** (10 min.) Write a Python iterator class capable of iterating over all of the items in your HashTable container, and update the HashTable class so it is an iterable class that uses your iterator class.

c) ** (1 min.) Write a for loop that iterates through your hash table using idiomatic Python syntax, and test this with both your class and generator.

d) ** (5 min.) Now write the loop manually, directly calling the dunder functions (e.g., __iter__) to loop through the items.

e) ** (5 min.) Finally, add a forEach() method to your HashTable class that accepts a lambda as its parameter, and applies the lambda that takes a single parameter to each item in the container:

```
ht = HashTable()
# add a bunch of things
ht.forEach(lambda x: print(x))
```

8. (10 min.) If you had to add generators to Brewin#, what language feature would you base the generator on, and what changes/additions would you have to make to that feature to make it support generators. What new language primitives (e.g., like `funccall`, `return`, etc.) would you have to expose to the programmer, if any?