Homework 7 – Spring 2023 (Due: May 24, 2023)

In this homework, you'll continue to explore parametric polymorphism as well as some of the fundamental OOP concepts.

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.) Discuss the benefits and drawbacks of (a) the template approach, (b) the generics approach, and (c) the duck typing approach for implementing generic functions/classes. Consider such things as: performance, compilation time, clarity of errors, code size, and type-checking (when does it occur - at runtime or compile-time). If you were designing a new language and had to pick one approach, which one would you choose and why?

Answer:

The template approach:

Benefits:

- Performance: Templates allow for compile-time polymorphism, which can lead to more efficient code execution at run-time. Why? The compiler generates specialized versions of the template function or class for each type used, resulting in optimized code.
- Earlier errors: Many template errors are reported during the compilation process rather than at run-time. helping developers identify and fix issues at an early stage.

Drawbacks:

- Code size: The use of templates can lead to code bloat. Each instantiation of a template with a different type creates a separate copy of the parameterized code, increasing the size of the compiled executable.
- Compilation time: Every time you use a templated function/class with a new type, the compiler generates a new version of that function/class and then compiles it.

So if you use a templated class/function with many types, this can significantly increase compilation time.

- More confusing errors: Since errors during compilation of generate on generated code (after the parameterized type(s) are applied to the template to yield a concrete version of the templated code), compilation errors can sometimes be more cryptic.

The generics approach:

Benefits:

- Code size: A generic is compiled once an a single object file is created, regardless of how many types the generic is parameterized with in the rest of the code base. This can reduce the overall size of the codebase relative to templates.
- Clarity of errors: Generic type errors are reported during the compilation process, providing clear error messages.
- Compilation time: Since a generic is compiled only once regardless of how many different types it's used with, this can improve compilation time relative to templates.

Drawbacks:

- Generics are generic: An unbounded generic is not able to leverage any operations that are specific to a particular type of object (e.g., asking an object to quack()). Its code must be compatible with all object types, using only operations like assignment. Even operations as simple as comparisons are banned in generics

unless they are bounded with a type. So this limits the capabilities of generic functions/classes.

- Performance: Bounded generics rely on runtime polymorphism, which can introduce performance overhead. Virtual function calls and type indirections may be necessary to dispatch to the appropriate implementation at runtime.

The duck typing approach:

Benefits:

- Flexibility: Duck typing is more flexible. I can use objects of entirely unrelated types with a function/class and the code can work without complex typing rules. This can simplify code and allow objects of different types to be used interchangeably if they support the required operations (e.g., quack())..
- Compile time: For compiled languages, compilation can be much faster since all error checking of duck typing occurs at runtime.

Drawbacks:

- Error detection: Errors are not detected until run-time which may complicate debugging a program.
- Performance: Duck typing incurs some runtime performance overhead due to dynamic method dispatch and type checking during execution.

One possible answer (inspired by ChatGPT): If I had to pick one approach, I would choose the generics approach. Generics strike a balance between code size and good performance while providing early type-checking during compilation. They allow for code reuse and provide clear error messages during the compilation process. While there may be some performance overhead compared to templates, the flexibility and ease of use offered by generics make them a suitable choice for most scenarios.

2. ** (5 min.) Explain why dynamically-typed languages can't support *parametric polymorphism*.

Answer:

Dynamically-typed languages are programming languages where variables are not explicitly assigned a type and a variable can thus refer to values of different types at runtime. Parametric polymorphism, on the other hand, requires that a function or class can be defined in such a way that it can have type(s) specified/parameterized for one or more variables used by that function or class (e.g., vector<int>, where int is the parameterized type) prior to execution. Since variables are typeless in dynamically-typed languages, it's impossible to assign parameterized types to variables, precluding the use of parametric polymorphism.

3. ** (5 min.) Explain how we can accomplish something like duck-typing in statically-typed languages that use templates. How is this similar to duck typing in dynamically typed languages, and how is it different (e.g., are there any pros/cons of either approach)?

Answer:

Since a template can be used to define a class/function that can operate on any type T (e.g, a duck or a doctor), so long as the operations (e.g., quack( )) used by that class/function on each templated variable are consistent with that type T, a single templated class/function can essentially operate on objects of completely different/unrelated types. This gives us duck-typing-like functionality:

```
template <typename T>
class MakeItQuack {
public:
  void process(T &v) {
    v.quack();
  }
};

class Duck {
public:
  void quack() { cout << "Quack!\n"; }
};

class Doctor {
public:
  void quack() { cout << "This cures insomnia!\n"; }
};

int main() {
  Duck daffy;
  MakeItQuack<Duck> quack1;
  quack1.process(daffy);   // causes a duck to quack

  Doctor fauchi;
  MakeItQuack<Doctor> quack2;
  quack2.process(fauchi);  // causes a doctor to quack
}
```

In the above example, MakeItQuack operates on Ducks and Doctors making them quack(), even though both classes are entirely unrelated.

Templates are similar to duck typing in that a single function/class can operate on many different types of variables that are unrelated, so long as those variables support the proper operations.

Templates are different than duck typing in that with templates, the compiler can validate that all operations performed by the templated class/function are all valid at compile time, since the statically-typed language can verify that every parameterized type (e.g., Nerd or Duck) used with the template supports all required operations used by the template (e.g., quack()). These compile-time errors can assist in writing correct code. In contrast, with duck typing, since variables have no types, type checking must occur at runtime at the time the operation (e.g., quack()) is performed, leading to more bugs/issues that can't be detected prior to execution.

4. (10 min.) Consider the following C++ container class which can hold pointers to many types of values:

```
class Holder {
private:
  static const int MAX = 10;
  void *arr_[MAX]; //  array of void pointers to objs/values
  int count_;
public:
  Holder() {
   count_ = 0;
  }
  void add(void *ptr) {
   if (count_ >= MAX) return;
   arr_[count_++] = ptr;
  }
  void* get(int pos) const {
   if (pos >= count_) return nullptr;
   return arr_[pos];
  }
};

int main() {
 Holder h;
 string s = "hi";
 int i = 5;
 h.add(&s);
 h.add(&i);

 // get the values from the container
 std::string* ps = (std::string *)h.get(0);
 cout << *ps << std::endl;  // prints: hi
 int* pi = (int *)h.get(1);
 cout << *pi << std::endl;  // prints: 5
}
```

This class does not use C++ templates, yet by using void pointers (void *) which are a generic type of pointer that can legally point to any type of value, it allows us to store a variety of (pointers to) values in our container object. This is how we used to do things before C++ added templates.

a. Discuss the pros and cons of the approach shown above vs. C++ templates.

Answer:

- pros:
  - this class can hold/work with any type of variable
- cons:
  - there's no type checking possible so the user could try to add/extract incompatible objects (e.g., add a string, then try to extract and treat that string as if it were a Dog object).
  - since the Holder class has no idea what type of objects/values it holds, or even if all the values it holds are of the same type, it's impossible for it to perform operations on those held objects (e.g., asking them to quack(), study(), etc.). With a template, since the parameterized type is known, the templated function/class can use any operations it likes so long as they're compatible with the templated type.

b. How is this approach similar or different to generics in languages like Java?

Answer:

In Java generics, the generic function/class can make no assumptions about the type of the variables used with that class (unless the generic is a bounded generic), and therefore it is also limited in the set of operations it can perform on values that are processed by the generic (like our C++ code above).

On the other hand, when we instantiate a generic (e.g., ArrayList<Dog> x = new ArrayList<Dog>), the compiler can at least type-check all operations that use x. For example it can verify that only Dog objects are added to x (e.g., x.add(new Dog("fido"));), and code that extracts an item from x doesn't try to treat the extracted object as anything but a dog, e.g., a String.

c. What change might we make to the above program to make it work more like a bounded generic class?

Answer:

We could change the code so instead of using a void *, it used another concrete type like a Mammal *. This would now limit the Holder class to processing only Mammal-related objects (People, Dogs, Raccoons), but also it would allow the code to call any methods that are present in a Mammal class or any of its superclasses (e.g., breathe(), have_live_births(), etc.)..

5. ** (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?

Answer:

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!
```

6. (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?

Answer (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.

7. ** (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?

Answer:

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.

8. ** (5 min.) Why aren't interfaces used/needed in dynamically-typed languages?

Answer:

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.

9. (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++?

Answer (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.

10. ** (5 min.) Give an example of where you'd prefer interface inheritance instead of traditional subtype inheritance. Why? Give an example of when you'd want to do the opposite.

Answer (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).