AP Computer Science Name: _____
Chapter 4 Notes (5)

FINAL WORDS ON POLYMORPHISM

Polymorphism is both simple and profound. Simply stated it just means that each robot (or object) does what it knows how to do when sent any message. However, the consequences of this are quite deep. In particular, it means that when you as a programmer send a message to a robot referred to by a reference, you don't know in general what will happen. You know the type of the reference, but perhaps not the type of the object it points to. You don't decide. The robot to which the reference points will decide. You might think you have a simple `UrRobot`, since the reference variable you use has that type, but the variable might point to a robot in some sub class of `UrRobot` instead. So even if you say something as simple as

```
myRobot.move();
```

you can't be sure what will happen. In particular, the following is legal

```
UrRobot mary = new MileWalker();
```

in which case `mary.move()` will move a mile, rather than a block. Combining this with the ability to write a method that has a parameter of type `UrRobot` to which you can pass any robot type, and again with the possibility of changeable strategies in robots, means that what happens when you write a message, may not be precisely determinable from reading the program, but only from executing it. Moreover, the same variable can refer to different robots of different types at different times in the execution of the program.

This doesn't mean that all is chaos, however. What it does mean is that you need to give meaningful names to methods, and also guarantee that when that kind of robot receives that message, it will do the right thing for that kind of robot. Then, when some programmers decide to use that kind of robot, they can be assured that sending it a message will do the right thing, even if they don't know precisely what that will be. The main means of achieving this is to keep each class simple, make its name descriptive of what it does, and make it internally consistent and difficult to use incorrectly.

Perhaps even more important are two rules: one for interfaces and one for inheritance. When you define and interface you need to have a good idea about what the methods defined there mean logically, even though you don't implement them. When you do implement the interface in a class, make sure that your implementation of the methods is consistent with our logical meaning. If you do this consistently, then the interface will have a logical meaning that a programmer can depend upon, even though they don't know what code will be executed.

The rule for inheritance is a little more precise. Think of inheritance as meaning specialization. If class `SpecialRobot` extends class `OrdinaryRobot`, for example, make sure that logically speaking a special robot is really just a specialized kind of ordinary robot. So it wouldn't make sense for an `IceSkaterRobot` to extend `Carpenter`, for example, since ice skaters are not specialized carpenters, but a different kind of thing altogether. If you do this faithfully, then the logic of your program will match the rules of assignment in the language. Our shorthand for this is called the IS-A rule. Ask "IS-A skater a carpenter?" The answer is no, so skater classes should not be subclasses of any `Carpenter` class. On the other hand a "`Carpenter` IS-A `UrRobot`" and has all of its capabilities and more, so it is appropriate to make `Carpenter` a subclass of `UrRobot`. If you create class and don't specify what it extends, then it automatically extends the built-in Object class.

You may have noticed that we have been careful to initialize all of our instance variables or fields when we declare them. This helps avoid problems.

Finally, we note that we have two different kinds of decomposition of programs. Stepwise refinement takes a complex method and breaks it into simpler methods. Polymorphism, on the other hand, takes related tasks and puts them into different objects.