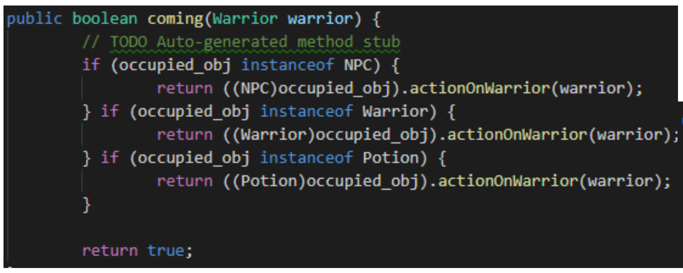
CSCI3180 assignment 2 – Task2 and Report

Name: Lun Yin Fung

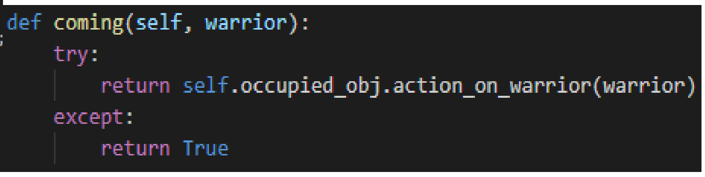
SID: 1155092566

1.

Code example of duck typing: Java(left) and Python(right)



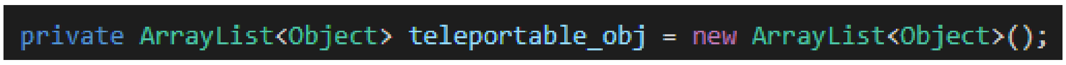
```java
public boolean coming(Warrior warrior) {
	// TODO Auto-generated method stub
	if (occupied_obj instanceof NPC) {
		return ((NPC)occupied_obj).actionOnWarrior(warrior);
	} if (occupied_obj instanceof Warrior) {
		return ((Warrior)occupied_obj).actionOnWarrior(warrior);
	} if (occupied_obj instanceof Potion) {
		return ((Potion)occupied_obj).actionOnWarrior(warrior);
	}

	return true;
```



```python
def coming(self, warrior):
    try:
        return self.occupied_obj.action_on_warrior(warrior)
    except:
        return True
```

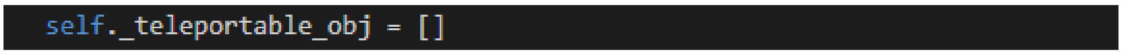
In the class inheritance design in task 4, Warrior and Portion is not the subclass of NPC, but they all have the instance method actionOnWarrior(). In Java(left), a statically typed language, we need to separate the flow of the instance method to three, we need to check the type of "occupied_obj" and then call the instance method with type casting. In Python(right), a dynamically typed language, we can just call the instance method of "occupied_obj" without type casting and checking its type (by some if statement) as the error will be caught (e.g. an object without instance method actionOnWarrior()). Thanks to duck typing, the logic flow of the method become clear in Python with less codes. Duck typing is possible in the above Python code fragment because the type of the variable is decided by its value and type checking is happened in runtime.

Code example of mixed type collection of data structure: Java(up) and Python(down)



```java
private ArrayList<Object> teleportable_obj = new ArrayList<Object>();
```

In the above Java code, teleportable_obj is ArrayList of type Object and only variable of type object can be added to this list (in the above code, Warrior and Potion can be added to the ArrayList but not integer etc.)

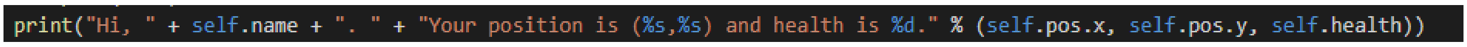


```python
self._teleportable_obj = []
```

In the above Python code, teleportable_obj is a without a type, and in fact, variable of different type, like integer or float or object type can also be added to this list without. For example, possible_list = [1,3.43], which cannot be declared in Java, or the statically typed language.

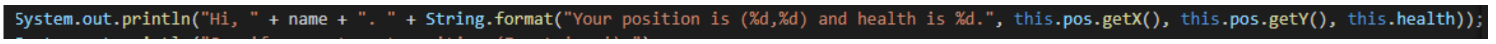
2.

First, the string formation in Python is better.



```python
print("Hi, " + self.name + ". " + "Your position is (%s,%s) and health is %d." % (self.pos.x, self.pos.y, self.health))
```

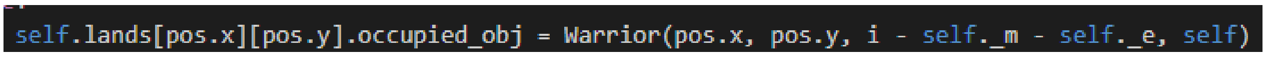
In the above Python code, we can use a "%" to assign the value of the placeholder in the string.



```java
System.out.println("Hi, " + name + ". " + String.format("Your position is (%d,%d) and health is %d.", this.pos.getX(), this.pos.getY(), this.health));
```

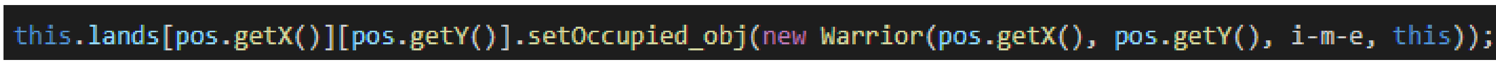
However, in the above Java code, it is required to use the "String.format" static method to assign the value of the placeholder, which make the code's readability lower. Moreover, in Java, "String" is considered as an object, thus we need to use the class method/instance method to format the String or get attribute from the String (usually the method name will be longer because of the package name).

Second, the getter and setter in Python is better than those in Java.



```python
self.lands[pos.x][pos.y].occupied_obj = Warrior(pos.x, pos.y, i - self._m - self._e, self)
```

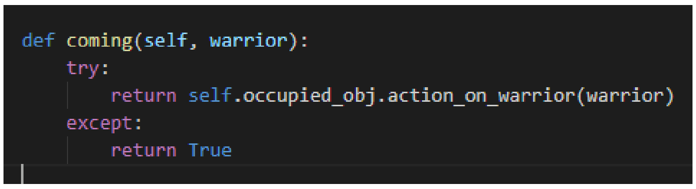
In the above Python code, with the help of @property and setter, we can treat the instance method as instance variable, we can call them without the parentheses, while the setter can also check the validity of the input value of the setter.



```java
this.lands[pos.getX()][pos.getY()].setOccupied_obj(new Warrior(pos.getX(), pos.getY(), i-m-e, this));
```

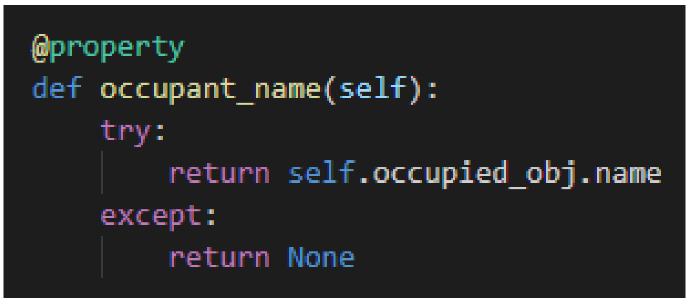
In the above Java code, there are lots of usage of setter and getter, which produce lots of parentheses, thus reducing the readability of the code. Besides, the programmer may be confused because of the parentheses and close the parentheses at wrong place accidentally.

3.

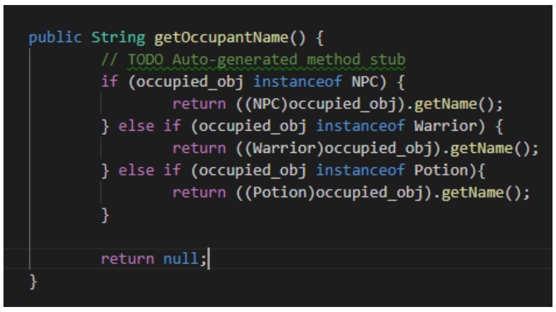


```python
def coming(self, warrior):
    try:
        return self.occupied_obj.action_on_warrior(warrior)
    except:
        return True
```

In the above code, duck typing is used. If "self" do not have the instance method actionOnWarrior, it will return true, which means the warrior can go and stay at the "self.pos" (i.e. only EMPTY lands do not have actionOnWarrior in task4, so it always returns true). For other objects' action on warrior, programmers can define instance method with same name and different action to those class react with Warrior. If a new class, such as "SWORD" is added to the game in the future, the programmers can define a new actionOnWarrior() instance method in "SWORD" without changing other part of the codes.



```python
@property
def occupant_name(self):
    try:
        return self.occupied_obj.name
    except:
        return None
```



```java
public String getOccupantName() {
        // TODO Auto-generated method stub
        if (occupied_obj instanceof NPC) {
                return ((NPC)occupied_obj).getName();
        } else if (occupied_obj instanceof Warrior) {
                return ((Warrior)occupied_obj).getName();
        } else if (occupied_obj instanceof Potion){
                return ((Potion)occupied_obj).getName();
        }

        return null;
}
```

In the above Java code(right), we need to check the class of the objects before calling the getName() instance method, although we all know that all these objects have a instance method called getName(). However, in Python(left) we can get the occupied_obj.name by calling the getter of occupant name, without considering the class, which increase the readability of the codes.