Q1)

```java
public class Warrior{
        private static final int HEALTH_CAP = 40;
        private Pos pos;
        private int index;
        private int health;
        private String name;
        private Map map;
        private int magic_crystal;
        public Warrior(int posx, int posy, int index, Map map) {
                this.pos = new Pos(posx, posy);
                this.index = index;
                this.map = map;
                // TODO Auto-generated constructor stub
                this.name = "W" + Integer.toString(index);
                this.health = HEALTH_CAP;
                this.magic_crystal = 10;
        }
```

```python
class Warrior():
    def __init__(self, posx, posy, index, mapp):
        self._pos = Pos(posx, posy)
        self._index = index
        self._map = mapp
        self._name = ("W%s" % str(index))
        self._health = HEALTH_CAP
        self._magic_crystal = 10
```

*As can be seen, the left Java code which does not support dynamic typing is extremely wordy and requires longer lines of code. However, the python code, which supports dynamic typing, is completely readable. Another advantage would be the ability to do duck-typing. For instance, on the left code, the instance “map” **must** be an instance of Map class. However, on the right, “mapp” can basically be any object.

Q2)

Scenario 1:

```java
public void setNumOfAliveMonsters(int numOfAliveMonsters) {
        this.numOfAliveMonsters = numOfAliveMonsters;
}
```

```python
@num_of_alive_monsters.setter
def num_of_alive_monsters(self, num_of_alive_monsters):
    self._num_of_alive_monsters = num_of_alive_monsters
```

*The getters and setter in Java require you to write a function and use it every time you need to excess it. But in python, when we define setters in the way shown in “num_of_alive_monsters” function, we can just invoke the object’s field and give it any value we want instead of calling a function. For ex. In java myMap.setNumOfAliveMonsters = 5; but in python my_map.num_of_alive_monsters = 5

Scenario 2:

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

        return true;
}
```

```python
def coming(self, warrior):
    if(self._occupied_obj != None):
        return self._occupied_obj.action_on_warrior(warrior)
    return True
```

*In java implementation, we have to do type-casting, type-checking etc. And this causes the programmer to write more codes and worry about more things, thus opening potential for more errors done by the programmer. While in the python case, the programmer does not need to worry about type-casting or type-checking as long as write objects are passed to the function. So the programmer saves more time, writes lesser lines of code and the programmer is prone to less mistakes because he does not have to worry about them.

Q3)

```java
public void teleportAll() {
        for (Object obj : teleportable_obj) {
            if (obj instanceof Warrior) {
                    ((Warrior) obj).teleport();
                }
                else if (obj instanceof Potion) {
                    ((Potion) obj).teleport();
                }
        }
}
```

```python
def teleport_all(self):
    global teleportable_obj

    for i in range(len(teleportable_obj)):
        if(teleportable_obj[i] != None):
            teleportable_obj[i].teleport()
```

*As can be seen, the above Java code which does not have duck-typing is very verbose while the Python code on the right is definitely more readable. This case will be more apparent as the number of object types that we have to check increases in the java program, but our code in our python function will not change and we won't have to include check cases for other newly introduced objects.