COMPSCI 230 Tutorial 4 - Answers

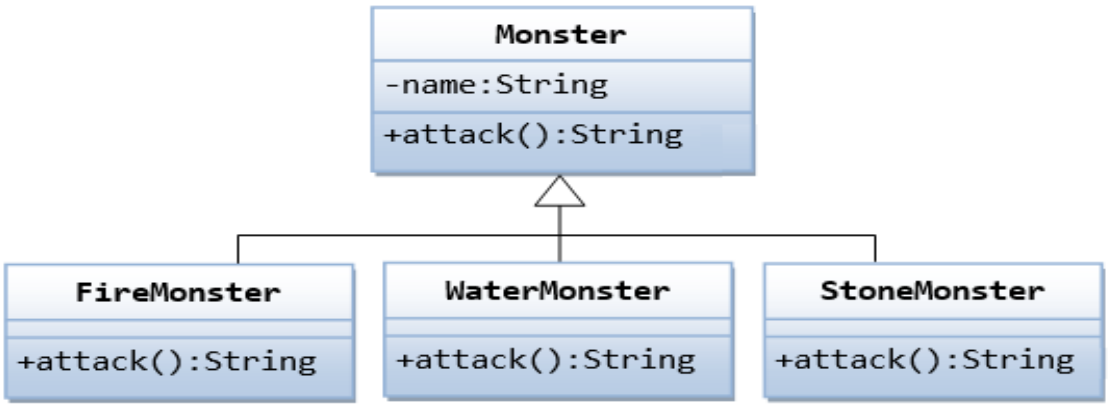
Java: Inheritance, Overriding, Polymorphism

Today we will look at overriding, polymorphism & abstraction, with a particular focus on inheritance. This tutorial will help you understand how to create class hierarchy.

1. Remind ourselves of what the following concepts mean:

Inheritance	**In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes. A class that is derived from another class is called a subclass (also a derived class, extended class, or child class). The class from which the subclass is derived is called a superclass (also a base class or a parent class).**
Abstract method	**An abstract method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body). You use the keyword abstract to declare an abstract method.**
Abstract class	**Any class that contains abstract methods must be declared as abstract class, as its implementation is not complete. You can use the keyword abstract to declare an abstract class. On the other hand, classes that have complete implementation are known as concrete classes.**
Overriding	**If a subclass implements its own version of a superclass method (same signature but different body) then that is called (method) overriding.**
Polymorphism	**When a superclass reference is used to refer to a subclass object.**

2. Let us consider a game app, where we have many types of monsters that can attack. We shall design a superclass called Monster and define the method attack() in the superclass. The subclasses shall then provide their individual implementation (overriding). In the main program, we declare references of superclass, substituted with instances of subclass; and invoke method defined in the superclass (polymorphism). Below UML diagram gives more details about the class hierarchy.



Superclass Monster.java

```
/*
 * The superclass Monster defines the expected common behaviours for its
subclasses.
 */
public class Monster {
	// protected instance variable
	protected String name;

	// Constructor
	public Monster(String name) {
		this.name = name;
	}

	// Define common behaviour for all its subclasses
	public String attack() {
		return "Err.. I don't know how to attack!";
		// We have a problem here!
		// We need to return a String; else, compilation error!
	}
}
```

Subclass FireMonster.java

```
public class FireMonster extends Monster {
	// Constructor
	public FireMonster(String name) {
		super(name);
	}

	// Subclass provides actual implementation
	@Override
	public String attack() {
		return name + " Attack with fire!";
	}
}
```

Subclass WaterMonster.java

```
public class WaterMonster extends Monster {
	// Constructor
	public WaterMonster(String name) {
		super(name);
	}

	// Subclass provides actual implementation
	@Override
	public String attack() {
		return name + " Attack with water!";
	}
}
```

Subclass StoneMonster.java

```
public class StoneMonster extends Monster {
	// Constructor
	public StoneMonster(String name) {
		super(name);
	}

	// Subclass provides actual implementation
	@Override
	public String attack() {
		return name + " Attack with stones!";
	}
}
```

MainClass.java

```
public class MainClass {
	public static void main(String[] args) {
		// Declare references of the superclass
		// and assign instances of subclasses.
		Monster m1 = new FireMonster("Fire Lion"); // upcast
		Monster m2 = new WaterMonster("Strike"); // upcast
		Monster m3 = new StoneMonster("Blizzard"); // upcast

		// Invoke the actual implementation
		System.out.println(m1.attack()); // FireMonster's attack()
		System.out.println(m2.attack()); // WaterMonster's attack()
		System.out.println(m3.attack()); // StoneMonster's attack()

		// m1 dies, generate a new instance and re-assign to m1.
		m1 = new StoneMonster("Metamorphic"); // upcast
		System.out.println(m1.attack()); // StoneMonster's attack()

		// We have a problem here!!!
		Monster m4 = new Monster("Green");
		System.out.println(m4.attack()); // garbage!!!
	}
}
```

3. Modify above class hierarchy to solve the problem we encountered when we created the instance of superclass Monster and attack() was called.

Hint: This can be resolved via abstract method and abstract class.

Superclass Monster.java

```
/*
 * This abstract superclass Monster contains an abstract method attack(),
to be implemented by its subclasses.
 */
abstract public class Monster {
	// protected instance variable
	protected String name;

	// Constructor
	public Monster(String name) {
		this.name = name;
	}

	// All Monster subclasses must implement a method called attack()
	abstract public String attack();
}
```

MainClass.java

```
public class MainClass {
	public static void main(String[] args) {
		// Declare references of the superclass
		// and assign instances of subclasses.

		Monster m1 = new FireMonster("Fire Lion"); // upcast
		Monster m2 = new WaterMonster("Strike"); // upcast
		Monster m3 = new StoneMonster("Blizzard"); // upcast

		// Invoke the actual implementation
		System.out.println(m1.attack()); // FireMonster's attack()
		System.out.println(m2.attack()); // WaterMonster's attack()
		System.out.println(m3.attack()); // StoneMonster's attack()

		// m1 dies, generate a new instance and re-assign to m1.
		m1 = new StoneMonster("Metamorphic"); // upcast
		System.out.println(m1.attack()); // StoneMonster's attack()

		// Cannot create instance of an abstract class
		Monster m4 = new Monster("Green"); // Compilation Error!!
		// System.out.println(m4.attack());// No more garbage!!!
	}
}
```