Tutorial 5

Question 1

Abstract Classes

- Convenient sometimes to define a class to represent a generic concept
- Common case - describing a category of classes such as Shapes
- Concrete members of that category have separate classes which inherit from the parent, abstract class
- We specify the fields and methods but do not necessarily provide an implementation

Example: Shapes

- What are properties and methods all shape classes should possess?
  - Color
  - getArea()
- Use inheritance - **Shape** as parent, add a field for **color**.
- What about getArea()? Impossible to calculate area of ‘generic’ shape.
  - Define getArea() as abstract

Question 2

Liskov Substitution Principle

- Motivation: If a type **Child** inherits / subtypes a type **Parent** (**Child < Parent**), then code written to operate on **Parent** should work for **Child** too.
- Principle: If **Child** subtypes **Parent**, **Parent** should be substitutable by **Child** without altering correctness, or expected behaviour

Question 3

Interfaces

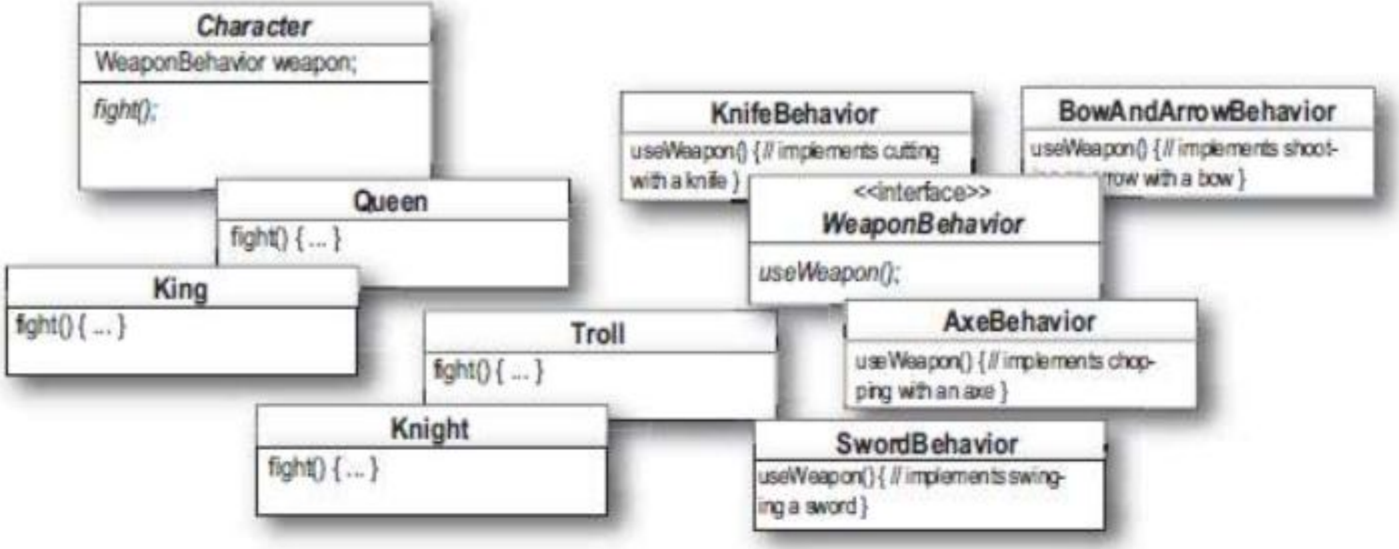
- Similar to abstract classes, but only contain abstract methods
- Useful when we want to describe a category of objects using their behaviors / operations only
- Or when we want to describe a trait / behavior which can be added onto different objects

Java: Interface vs Abstract Class

- Interfaces contain **methods only -** no fields or constructor like Abstract Class
- Interface methods cannot be private or protected

- A Class can implement **multiple interfaces**, whilst a Class can only inherit from one **Abstract Class**
- Important for creating behavior interfaces such as Iterable or Cloneable

Question 4



Character
WeaponBehavior weapon;
fight();

Queen
fight() { ... }

King
fight() { ... }

Troll
fight() { ... }

Knight
fight() { ... }

KnifeBehavior
useWeapon() { // implements cutting with a knife }

BowAndArrowBehavior
useWeapon() { // implements shoot- row with a bow }

<<interface>>
WeaponBehavior
useWeapon();

AxeBehavior
useWeapon() { // implements chopping with an axe }

SwordBehavior
useWeapon() { // implements swinging a sword }

```
setWeapon(WeaponBehavior w) {
    this.weapon = w;
}
```

Admin

Project Stuff

- Please add all teammates to your GitHub team and create a repo using the GitHub/CSE website
- User Stories are due next week - will be checking them and giving feedback
- Create user stories using the GitHub Projects Task Board
- Make sure you follow the R-G-B (Role - Goal - Benefit)
- Create Epics and break them down into smaller user stories
- Define acceptance criteria

Assignment 1

- Marking will begin next week (after late submissions / special consideration), should be done in 2 weeks

Variance

- Variance is a concept in OOP which relates to how subtyping of simple components affects subtyping of complex / composite types
- Covariance - composite type preserves ordering of component types
- Contravariance - composite type reverses ordering of component types

Variance - Arrays

- Suppose we have an **Animal** class, and a **Cat** class: **Cat < Animal**

- If Arrays (the composite) were covariant: an array of Cats is an array of Animals (**Cat[] < Animal[]**)
- If Arrays were contravariant: an array of Animals is an array of Cats (**Animal[] < Cat[]**)
- If Arrays were invariant: an array of Cats not an array of Animals, an array of Animals is not an array of Cats
- In Java, arrays are **covariant**

Variance - Return Types

- Suppose we have an **AnimalShelter** class, which has a list of **Animals**
- It has two methods - **adopt** (removes animal) and **rescue** (inserts animal)
- We then create a **CatShelter**: **CatShelter < AnimalShelter**