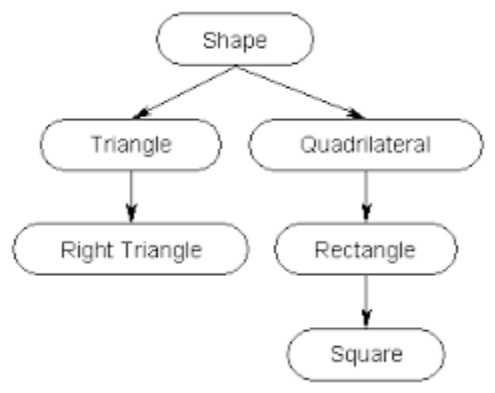
Recitation 12

Inheritance



Overview

- HashMap (Via Bucket Sort algorithm)
- Inheritance
  - What is inheritance?
  - The reserved words super, extends and protected
  - Override
  - Abstract classes + Methods
  - Inheritance Vs Composition.
- Interface

Recitation 12

HashMap

Bucket Sort

- Bucket Sort is a sorting algorithm that divides the input array into several groups called buckets and then sorts each bucket individually. It is particularly useful for uniformly distributed data over a range.

Bucket Sort - Steps

- Initialize Buckets:
  - Create a `HashMap<Integer, LinkedList<Double>>` for the buckets.
  - Each bucket is a LinkedList, allowing for efficient insertion and removal.
- Distribute Elements:
  - Each element is mapped to a bucket index using the formula:
  - `int bucketIndex = (int) (num * n);`
  - This assumes all numbers are within the range [0, 1).
- Sort Individual Buckets:
  - Each non-empty bucket is sorted using a helper function sortBucket, which employs Insertion Sort.
- Concatenate Sorted Buckets:
  - The contents of all buckets are concatenated back into the original array in order.

Bucket Sort - Implementation

```
public static void bucketSort(double[] arr) {
        if (arr == null || arr.length <= 1) {
            return;
        }

        // Step 1: Create buckets using HashMap
        int n = arr.length;
        HashMap<Integer, LinkedList<Double>> buckets = new HashMap<Integer, LinkedList<Double>>();

        // Initialize buckets as LinkedLists
        for (int i = 0; i < n; i++) {
            buckets.put(i, new LinkedList<Double>());
        }
        // Step 2: Distribute elements into buckets
        for (double num : arr) {
            int bucketIndex = (int) (num * n); // Assuming numbers are in range [0, 1)
            buckets.get(bucketIndex).add(num);
        }
        // continue in next slide
}
```

Bucket Sort - Implementation

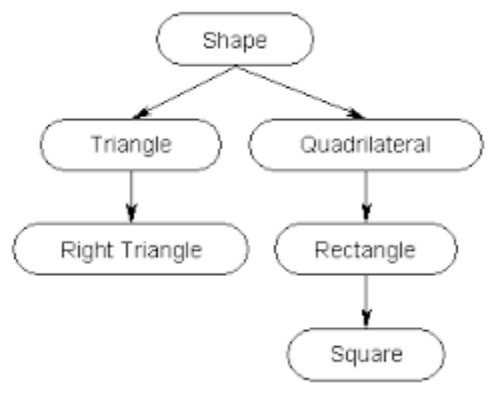
```
public static void bucketSort(double[] arr) {
        // Step 3: Sort individual buckets using insertion sort
        for (int i = 0; i < n; i++) {
            LinkedList<Double> bucket = buckets.get(i);
            if (bucket != null && bucket.size() > 1) {
                sortBucket(bucket);
            }
        }
        // Step 4: Concatenate all buckets back into the original array
        int index = 0;
        for (int i = 0; i < n; i++) {
            LinkedList<Double> bucket = buckets.get(i);
            if (bucket != null) {
                for (double num: bucket) {
                    arr[index++] = num;
                }
            }
        }
}
```

Bucket Sort - Implementation

```
// Helper function to sort a LinkedList using insertion sort
private static void sortBucket(LinkedList<Double> bucket) {
        for (int i = 1; i < bucket.size(); i++) {
            double key = bucket.get(i);
            int j = i - 1;

            // Move elements of bucket[0..i-1], that are greater than key, one position ahead
            while (j >= 0 && bucket.get(j) > key) {
                bucket.set(j + 1, bucket.get(j));
                j--;
            }
            bucket.set(j + 1, key);
        }
}
```

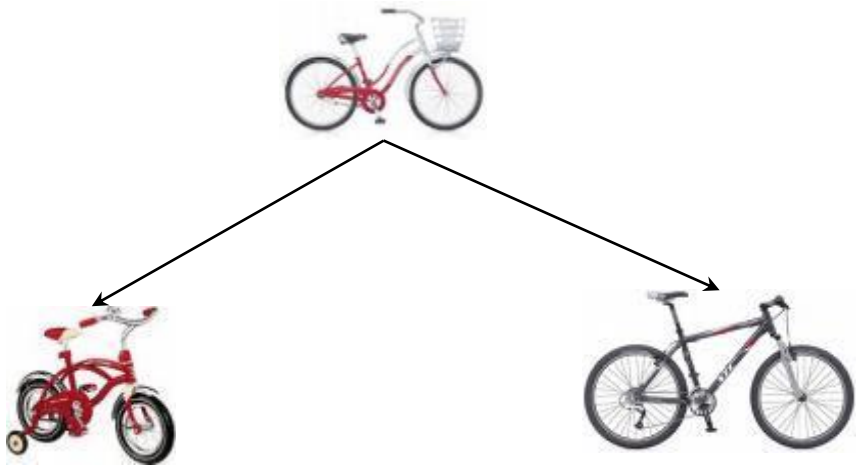
Recitation 12

Inheritance



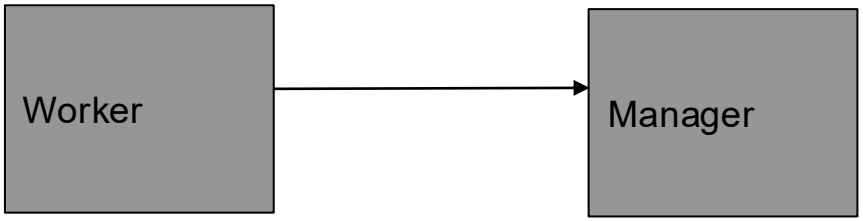
Inheritance

- We can think of a class as describing the state and behavior of some object.
- But some objects have hierarchy between them
- With inheritance we can define a new class based on an existing class class.
- Example:
  - Suppose we have a Bicycle class. Each such bicycle is defined by two Wheels, a Seat and Pedal fields.
  - We can now use this class to construct new classes:
    - Mountain Bikes.
    - Racing Bikes.
    - Etc.



Terminology

- Assume the class "Manager" inherits from "Worker"
- The "Worker" class is called the "super" class of "Manager"
- The "Manager" class is called the "sub"/"derived" class of "Worker"



Why do we need Inheritance ?

- **Reusability**: Inheritance allows for code reuse by allowing a subclass to inherit the data and behaviors of a superclass, thus reducing the need for redundant code. This is useful when extending the features of a class.
- **Extensibility**: Inheritance also allows for extensibility, as it facilitates the process of adding new features to a class rather than having to create a new class from scratch.
- **Maintainability**: When you have a parent class with multiple child classes, updating a shared feature is simple because you only need to make the change in the parent class. This is much more efficient than updating each class individually if there is no inheritance.

Inheritance

- A derived class inherits all **non-private methods and fields** of the super class except the constructors. Explicit calls must be made as the first line of the sub class's constructor method, with the **super** word, behaves similarly to the reserved word **this**. While the reserved word "this" is optional in most cases and can be called to refer to this object.
- We can define new methods/fields for the derived class based on the behavior of the derived class
- Inheritance describes a relation of "**is a**".
  - A mountain bike **is a** bicycle.
  - A racing bike **is a** bicycle.
- In other words, we can say that if B is a type of A, but A is not always B, then class B inherits from class A.
- If class B extends class A, then each instance of B contains inside it an instance of A.
- An instance of B is also an instance of A.

```
Bicycle b = new MountainBicycle(); //OK
```

- The other direction is not true;

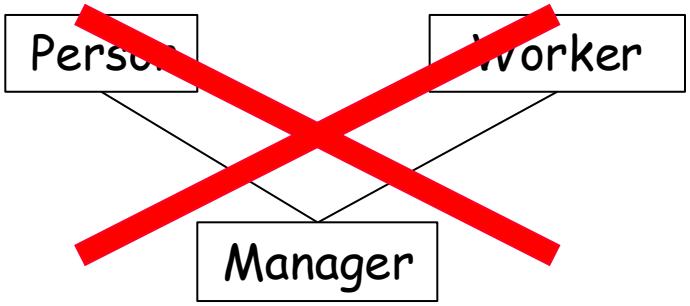
```
MountainBicycle b = new Bicycle(); //error
```

Reserved words

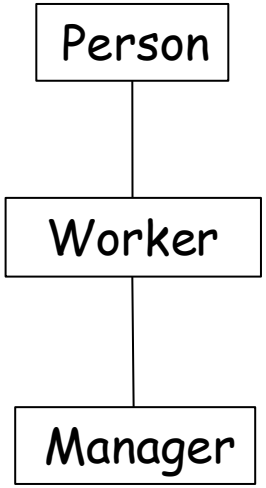
- In practice, we can say that one class inherit from another class it must use the reserved word "**extends**" followed by the name of the other class.
- The reserved word "**super**" allows us to call from the sub-class to its super-class.
- **"protected"** – is a reserved word, which declared privacy settings, if a field has a protected visibility modifier, this means it is visible only to subclasses of this class.

Inheritance – Rules

- A class can inherit from one class only.

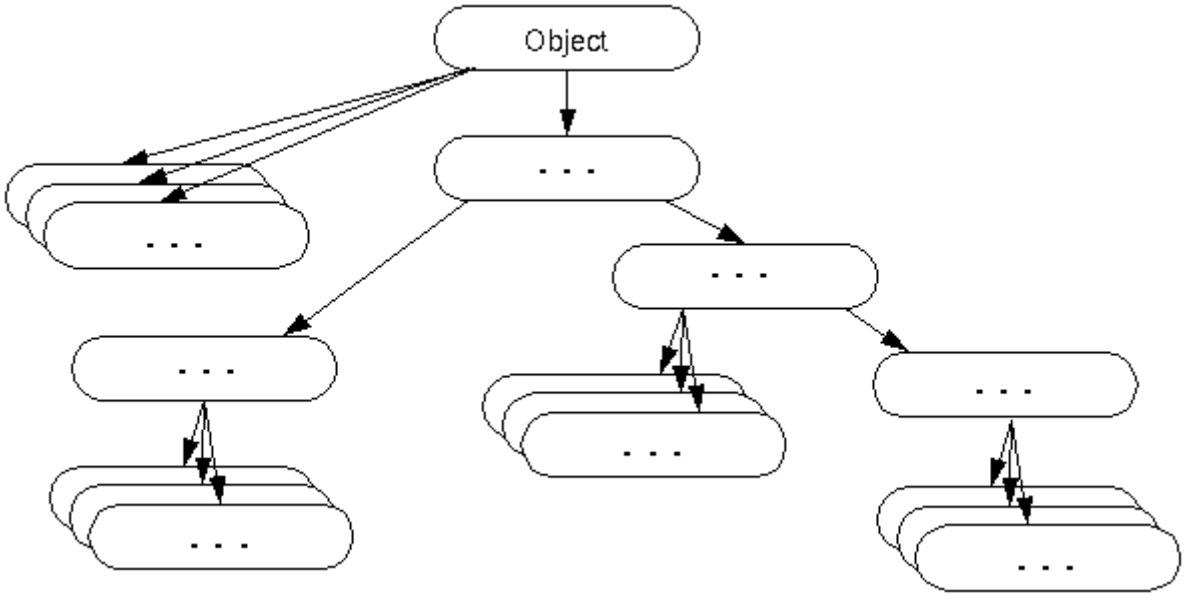


- Subclasses may still be derived though.



Inheritance – Rules

- All classes automatically inherit from Object class.
- You can think about Object as the most basic non-primitive data type in Java.
- Object already has some methods defined for it.
- The methods, equals(Object obj) and toString() are two such examples.
- So even if you didn't add those methods to your class, you may still use them.
- The only exception to the rule is when you use abstract classes, and methods



Recitation 12

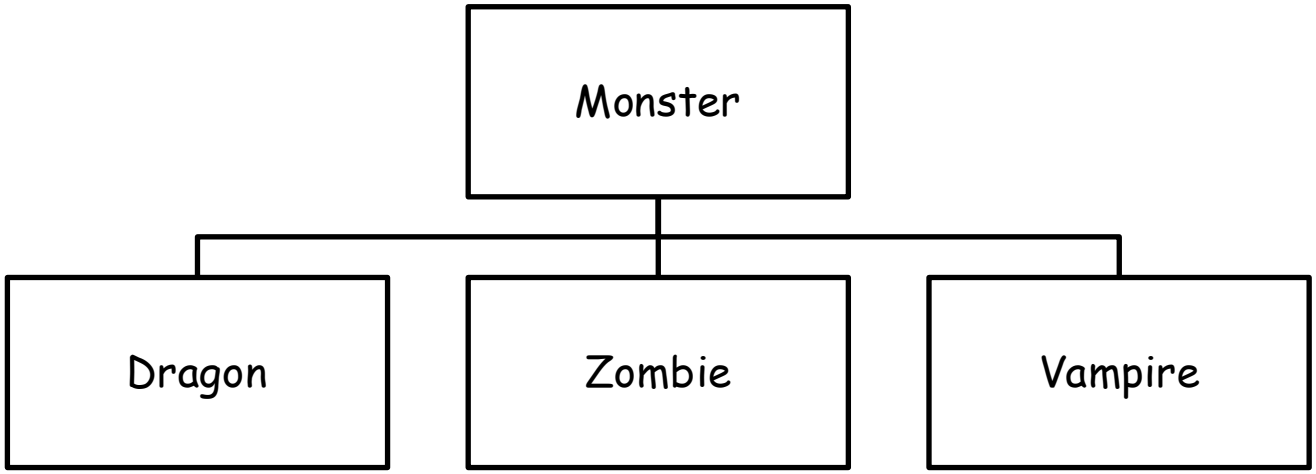
RPG Game

RPG game

- Imagine you are designing a fantasy role-playing game where different types of creatures, such as dragons, vampires, and zombies which battle against each other.
- Each creature has unique characteristics (e.g., hit points, strength) and special abilities, like breathing fire, healing themselves, or dodging attacks.
- You want to structure your code efficiently so that all creatures share common traits but can also have their own distinct behaviors.
- How would you design the relationships between these creatures' using inheritance and abstraction in Java?
- Specifically, consider how you would define a base class, extend it to create specific creatures, and implement unique behaviors while keeping your code reusable and scalable.

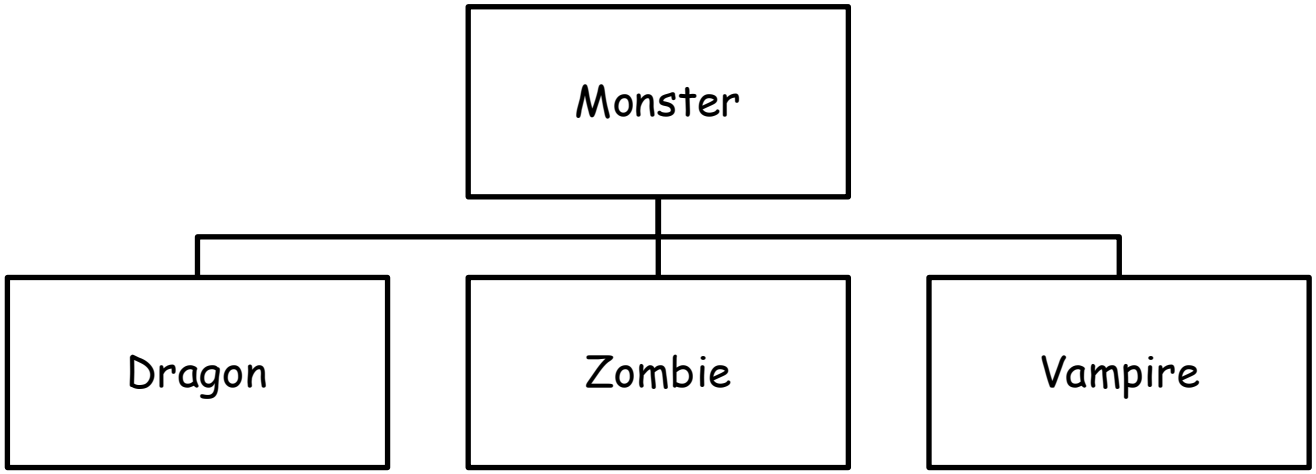
Plan

- Since we can say that all the creatures above are Monsters and share some sense of characteristics and methods, we can build a "Monster" class that will be a common ancestor. Then we will do some specification for them.
- We can say all Monsters should have hitPoints, strength, and name, they should be able to attack and take damage.
- While the other may have some behaviors and properties.



Step 1 – Create constructors and getters

- For the first step, we will create the constructors, fields and getters for each one.
- All monsters share common traits (fields): name, hit points, and strength. The last 2 should be positive numbers.
- Dragons will also have fire damage (should be positive).
- Zombies will also have poison damage (should be positive).



Example - Super Class

```
public class Monster {
    protected String name;
    protected int hitPoints;
    protected int strength;

    public Monster(String name, int hitPoints, int strength) {
        if (hitPoints <= 0 || strength <= 0){
           throw new IllegalArgumentException();
        }
        this.name = name;
        this.hitPoints = hitPoints;
        this.strength = strength;
    }
    public String getName() {
        return this.name;
    }
    public int getHitPoints() {
        return this.hitPoints;
    }
    public boolean isAlive() {
        return this.hitPoints != 0;
    }
    public int getStrength() {
        return this.strength;
    }
}
```

Example - Sub Class I

```
public class Dragon extends Monster {
    protected int fireDamage;

    public Dragon(String name, int hitPoints, int strength, int fireDamage) {
        super(name, hitPoints, strength);
        if (fireDamage <= 0){
         throw new IllegalArgumentException();
        }

        this.fireDamage = fireDamage;
    }

    public int getFireDamage() {
        return this.fireDamage;
    }
}
```

Example - Sub Class II

```
public class Zombie extends Monster {
    protected int poisonDamage;

    public Zombie(String name, int hitPoints, int strength, int poisonDamage) {
        super(name, hitPoints, strength);
        if (poisonDamage <= 0){
         throw new IllegalArgumentException();
        }

        this.poisonDamage = poisonDamage;
    }

    public int getPoisonDamage() {
        return this.poisonDamage;
    }

}
```

Example - Sub Class III

```
public class Vampire extends Monster {

    public Vampire(String name, int hitPoints, int strength) {
        super(name, hitPoints, strength);
    }
    // Question: Vampire doesn't have any additional fields, why is in a sub class?


}
```

Step 2 – Overriding

- Derived class can also override methods of the super class. In other words, the behavior of the sub class can change.
- Override is re-implementing a method in the sub-class, which already exists in the parent class.
- The action can either rewritten completely, or by depending the way the method operate in the super class.
- Let's go back to our Monsters and implement the method and takeDamage(int damage).
- The method get the amount of damaged to be subtracted from the hitPoints. When a Monster reaches 0 it is considered defeated. Meaning we don't want a negative number.

- We will add a characteristics to our classes
- Any damage Dragons receives is cut in half.
- Vampire will have 25% odds to dodge the attack.

Example - Super Class

```
public class Monster {
    protected String name;
    protected int hitPoints;
    protected int strength;

    public int takeDamage(int damage) {
        int dmgTaken = Math.min(damage, this.hitPoints);
        this.hitPoints -= dmgTaken;
        return dmgTaken;
    }
}
```

Example - Sub Class I

```
public class Dragon extends Monster {
    private int fireDamage;

    @Override
    public int takeDamage(int damage) {
          int dmgTaken = Math.min(damage / 2, this.hitPoints);
          this.hitPoints -= dmgTaken;
          return dmgTaken;
    }
}
```

Example - Sub Class II

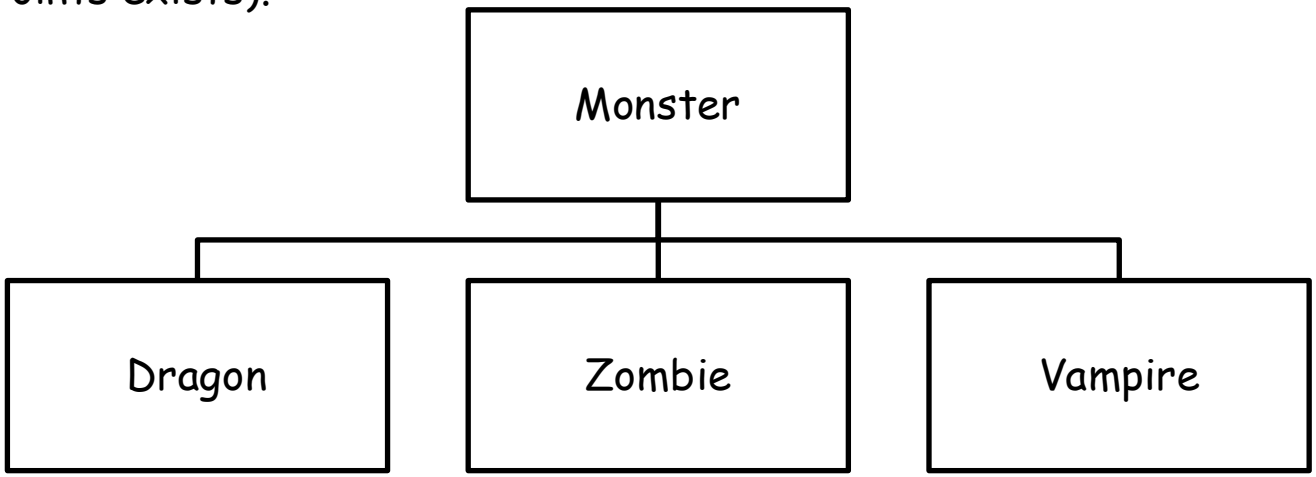
```
public class Zombie extends Monster {
    private int poisonDamage;

    // Question: what will happen if a Zombie will take damage?

}
```

Example - Sub Class III

```
public class Vampire extends Monster {

    @Override
    public int takeDamage(int damage) {
         if (this.didDodge()){
            return 0;
         }
         return super.takeDamage(damage);
    }

    public boolean didDodge() {
         return Math.random() < 0.25;
    }
}
```

Step 3 – Create abstract methods

- Each Monster can attack differently, by default we want that each monster will define its own attack, hence we don't want to define anything in "Monster" class.
- Furthermore, we don't want to define any generic Monster, we want every monster to act differently. So, a simple solution is turning Monster in "abstract" class and make "attack" into abstract method, to enforce overriding in all sub classes.
- All Monsters will use their strength as base to their attack, if they have special powers (fire, poison) it will be accumulated.
- Zombies will have 50% odds of not attacking, if they do attack, they will also deal poison damage.
- Dragons will also inflict fire damage.
- Vampires will heal 1/3 of damage of the damage taken to their hitPoints (assume no maxHitPoints exists).



Example - Super Class

```
public abstract class Monster {
    protected String name;
    protected int hitPoints;
    protected int strength;

    public abstract int attack(Monster other);
}
```

Example - Sub Class I

```
public class Dragon extends Monster {
    private int fireDamage;

    @Override
    public int attack(Monster other) {
         int res = other.takeDamage(this.strength + this.fireDamage);
         return res;
    }

}
```

Example - Sub Class II

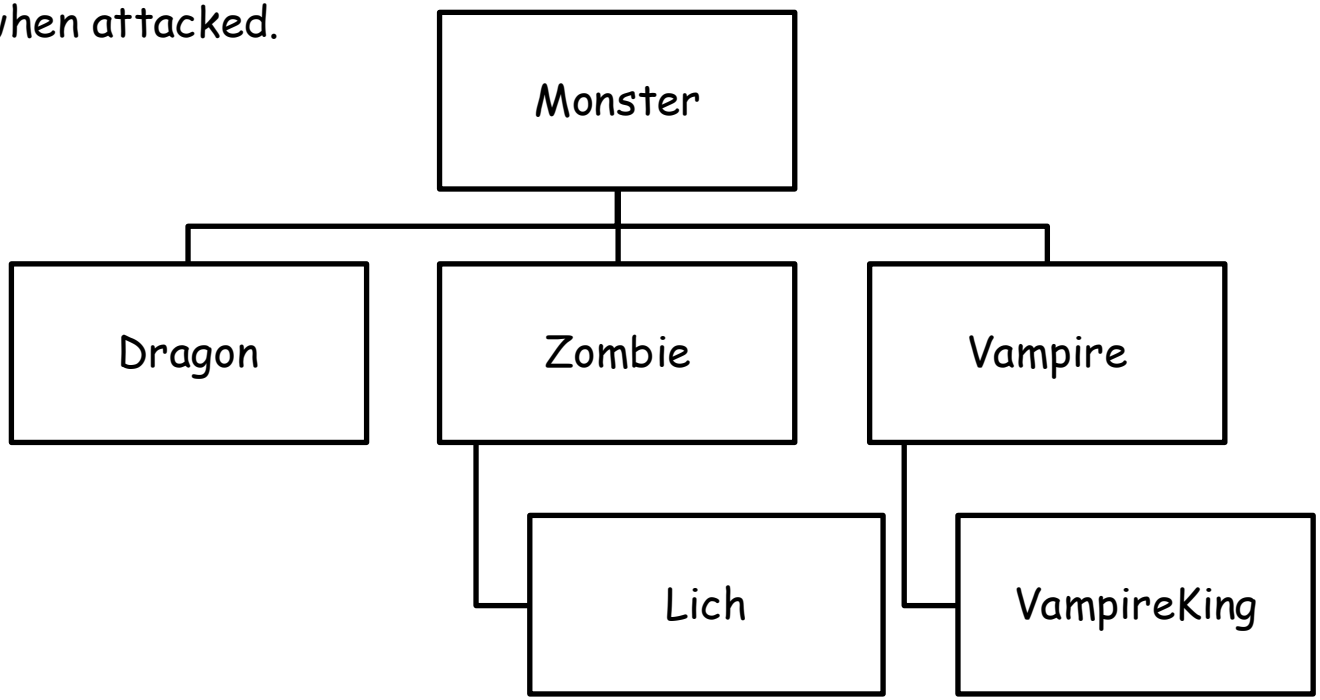
```
public class Zombie extends Monster {
    private int poisonDamage;

    @Override
    public int attack(Monster other) {
         if (this.didAttack()){
             int res = other.takeDamage(this.strength + this.poisonDamage);
             return res;
         }
         return 0;
    }
    public boolean didAttack() {
         return Math.random() < 0.5;
    }
}
```

Example - Sub Class III

```
public class Vampire extends Monster {

    @Override
    public int attack(Monster other) {
         int hp = other.takeDamage(this.strength);
         this.heal(hp);
         return hp;
    }

    public void heal(int hitPoints) {
         this.hitPoints += hitPoints / 3;
    }
}
```

Step 4 - Inheritance - Subclass is a super of other?

- Derived class can also be a super class another one.
- We will create 2 more classes, Lich (Zombie mage), VampireKing (The King of Vampires).
- The VampireKing will have 20% to attack twice in a row, and 43.75% odds to dodge the attack. (try to dodge twice)
- Lich will also have "mana" (magic power, positive number) and will be able to do magic attacks will have 80% to use a magic attacks and if it has "mana". The spell will deal 1-6 damage + strength + poison, and will decrease the mana by 1, when the Lich is not doing magic will act the same as Zombie. Furthermore, Lich will have 25% odds to dodge when attacked.



Example - Sub Class IV

```
public class VampireKing extends Vampire {
    public VampireKing(String name, int hitPoints, int strength) {
        super(name, hitPoints, strength);
    }

    @Override
    public int attack(Monster other) {
          int res = super.attack(other);
          if (this.didDoubleAttack()){
              res += super.attack(other);
          }
          return res;
    }

    @Override
    public int takeDamage(int damage) {
          if (super.didDodge()){
              return 0;
          }
          return super.takeDamage(damage);
    }

     public boolean didDoubleAttack (){
          return Math.random() < 0.2;
    }
}
```

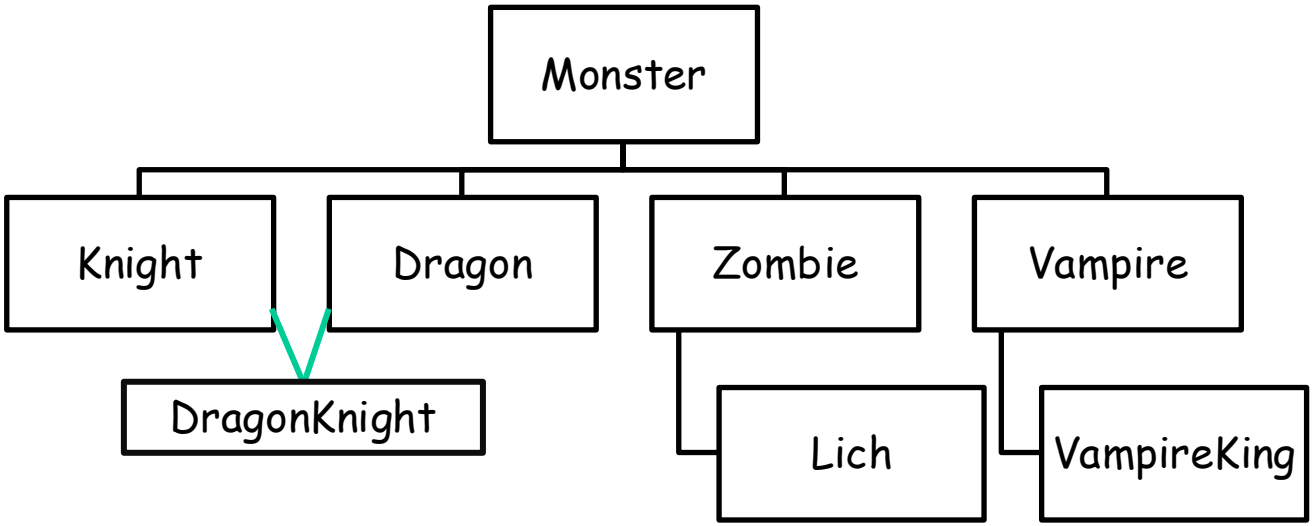
Example - Sub Class V

```
public class Lich extends Zombie {
   private int mana;
   public Lich(String name, int hitPoints, int strength, int poisonDamage, int mana) {
        super(name, hitPoints, strength, poisonDamage);
        if (mana < 0){
                throw new IllegalArgumentException();
        }
        this.mana = mana;
   }

   @Override
   public int attack(Monster other) {
       return this.didMagicAttack() ? this.magicAttack(other) : super.attack(other);
   }

   @Override
   public int takeDamage(int damage) {
       if (this.didDodge()){
          return super.takeDamage(damage);
       }
       return 0;
    }

    public int magicAttack(Monster other) {
          this.mana -= 1; // Note: will enter here only when there is mana (see attack & didMagicAttack)
          return other.takeDamage(this.spell());
     }

    public int spell() {
         return (int) ((Math.random () * 6 + 1) + super.getStrength() + super.getPoisonDamage());
    }

    public boolean hasMana(){
         return this.mana > 0;
    }

    public boolean didMagicAttack() {
         return Math.random() < 0.8 && this.hasMana();
    }

    public boolean didDodge() {
         return Math.random() < 0.25;
    }
}
```

Composition Vs Inheritance

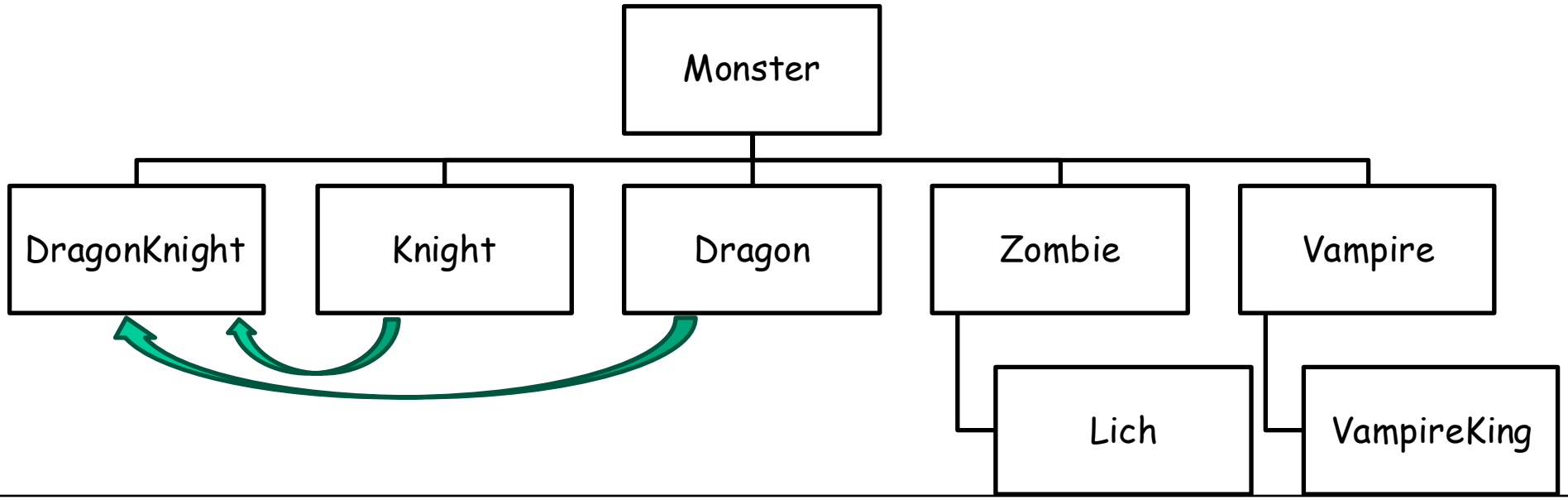
- As mentioned earlier, inheritance describes a relation of "**is a**", there are some cases this is not the case. For example, a Wheel is not a Car, but there should be some kind of relations between those objects.
- Most of them fall under the relation "**A has a B**", for example: Car has a Wheel, in those cases, we will use composition. Which means that class B will be a field of class A.
- Let's build 2 classes, Knight, and DragonKnight (Knight who rides a Dragon).

- Knight will be inheriting from Monster, and will have a defense, when taking damage, it will be reduced by that defense. When it attacks it will have 10% odds of critical hitting, which will take double the damage.
- DragonKnight will be composed from Knight and Dragon.
- Will DragonKnight be a Monster based on this definition?
- No! Since it is composed from Monsters not inherits from it,
- Is it a problem? Yes! Since it's not inheriting from Monster, we can't put a DragonKnight as a Monster to defeat..



Composition Vs Inheritance

- To fix it we will extend the name will be the combination of names of knight and dragon, the hitPoints will be the minimum of Knight and Dragon.
- When it attacks both the knight and the dragon will deal damage (including the dragon fire), when it is dealt damage, we will deal apply both the defense mechanisms of Knight (subtract by defense field) and Dragon (divide by 2). If one is out of hitPoints both hps will be changed to 0;



Example - Sub Class VI

```
public class Knight extends Monster {
    private int defense;
    private TitleEnum title;

    public Knight (String name, int hitPoints, int strength, int defense, TitleEnum title){
        super(name, hitPoints, strength);
        this.defense = defense;
        this.title = title;
    }

    public int getDefense(){
        return this.defense;
    }

    @Override
    public int takeDamage(int damage) {
        int damageTaken = (damage - this.defense);
        damageTaken = super.takeDamage(damageTaken);
        return damageTaken;
    }

    @Override
    public int attack (Monster other){
        int res = 0;
        if (this.isCriticalHit()){
            res = other.takeDamage(this.strength * 2);
        } else {
            res = other.takeDamage(this.strength);
        }
        return res;
    }

    public boolean isCriticalHit(){
        return Math.random() < 0.1;
    }

}
```

```
public enum TitleEnum {
    SIR("Sir"),
    LADY("Lady");

    private final String title;

    public TitleEnum(String title){
       this.title = title;
    }

    public String getTitle(){
        return this.title;
    }

}
```

Example - Sub Class VII

```
public class DragonKnight extends Monster {
    private Knight knight;
    private Dragon dragon;

    public DragonKnight(Knight knight, Dragon dragon){
        super(knight.getName() + " the brave and " + dragon.getName(),
                Math.min(knight.hitPoints, dragon.hitPoints), knight.getStrength() + dragon.getStrength());
        this.knight = knight;
        this.dragon = dragon;
    }

    @Override
    public int takeDamage(int damage) {
        int damageTaken = (damage - knight.getDefense()) / 2;
        damageTaken = super.takeDamage(damageTaken);
        this.knight.takeDamage(damage);
        this.dragon.takeDamage(damage);
        return damageTaken;
    }

    @Override
    public int attack(Monster other) {
        int res = this.knight.attack(other);
        res += this.dragon.attack(other);
        return res;
    }

    public boolean isAlive() {
        return this.hitPoints == 0 && this.knight.isAlive() && this.dragon.isAlive();
    }
}
```

Interface

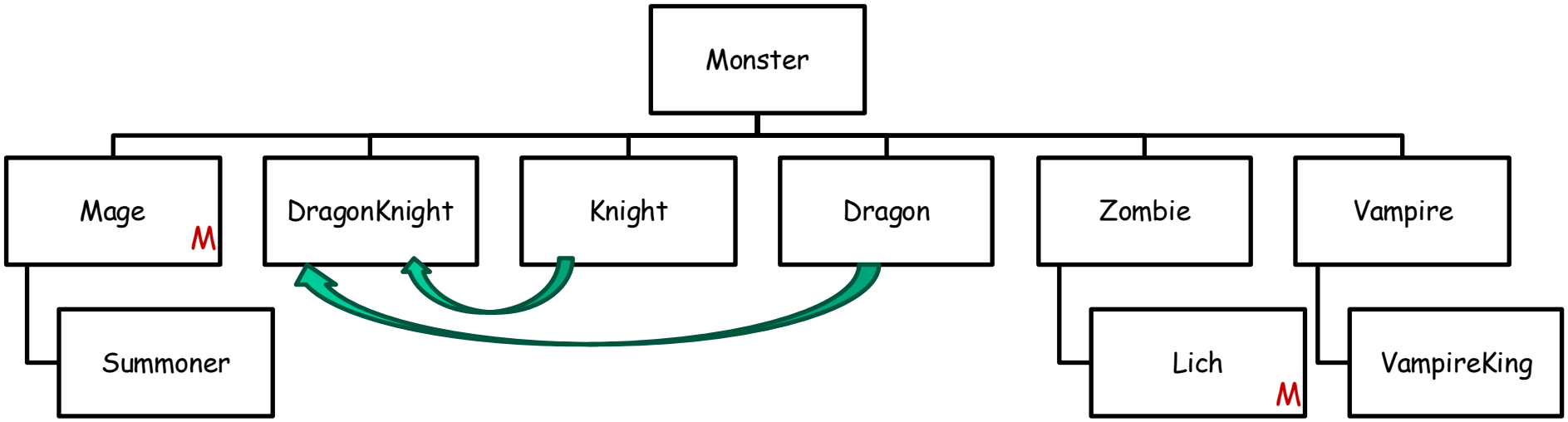
- Interface are another type of object, those are used for designing common properties and operations (behavior) that each member who wants to use this interface.
- Unlike Class inheritance we use the reserved word "implements" to tell what interface an object is inherits from.
- Furthermore, a class can inherit from multiple interfaces at once.
- A given Class can both inherits from both class (extends) and interface (implements).
- In an interface all properties are given as final

Composition Vs Inheritance Vs Interface

- As mentioned earlier, inheritance describes a relation of "**is a**", there are some cases this is not the case. For example, a Wheel is not a Car, but there should be some kind of relations between those objects, and composition, fall under the relation "***A has a B***", for example: Car has a Wheel, in those cases, we will use composition. Which means that class B will be a field of class A.
- Interfaces are all abstract in their nature but unlike abstract classes they dont follow the "is a" rule, if an object fall under the category **"A can do B"**. then A implement B, and B is an interface.

Implementing

- We will create an interface called "Magical" that interface will include 4 different methods, castSpell(Monster monster), getMana(), hasMana(), restoreMana().
- We will edit our Lich to implement the "Magical" interface and create a Classes "Mage" who extends Monster and implements Magical, and create a "Summoner" that extends Mage.
- Mage will cast spell if has mana and 80% of the times.
- Summoner will be able to summon a dragon for 2 mana.



Example – Interface

```
public interface Magical {

    int castSpell(Monster target);

    int getMana();

    boolean hasMana();

    void restoreMana(int amount);
}
```

Example – Interface with Subclass V

```
public class Lich extends Zombie implements Magical {
   private int mana;
   public Lich(String name, int hitPoints, int strength, int poisonDamage, int mana) {
        super(name, hitPoints, strength, poisonDamage);
        if (mana < 0){
              throw new IllegalArgumentException();
        }
        this.mana = mana;
   }
   // in addition to the code already done
    …
    @Override
    public int castSpell(Monster target) {
        if (!this.hasMana()) {
            return 0;
        }
        this.mana -= 1;
        return target.takeDamage(this.spell());
    }

    @Override
    public int getMana() {
        return this.mana;
    }

    @Override
    public boolean hasMana() {
        return this.mana > 0;
    }

    @Override
    public void restoreMana(int amount) {
        this.mana += amount;
    }
}
```

Example – Sub Class VIII

```
public class Mage extends Monster implements Magical {
    protected int mana;
    protected int spellPower;

    public Mage(String name, int hitPoints, int strength, int mana, int spellPower) {
        super(name, hitPoints, strength);
        if (mana < 0 || spellPower <= 0) {
            throw new IllegalArgumentException();
        }
        this.mana = mana;
        this.spellPower = spellPower;
    }

    @Override
    public int castSpell(Monster target) {
        if (!this.hasMana()) {
            return 0;
        }
        this.mana -= 1;
        int damage = this.spellPower + (int)(Math.random() * 6 + 1);
        return target.takeDamage(damage);
    }

    @Override
    public int getMana() {
        return this.mana;
    }

    @Override
    public boolean hasMana() {
        return this.mana > 0;
    }

    @Override
    public void restoreMana(int amount) {
        this.mana += amount;
    }

}
```

```
public class Mage extends Monster implements Magical {
    // Continue

    @Override
    public int attack(Monster other) {
        if (this.shouldCastSpell() && this.hasMana()) {
            return this.castSpell(other);
        }
        return other.takeDamage(this.strength);
    }

    public boolean shouldCastSpell() {
        return Math.random() < 0.8;
    }

    public int getSpellPower() {
        return this.spellPower;
    }


}
```

Example – Sub Class VIIII

```
public class Summoner extends Mage {
    private Dragon [] summonedDragons;
    private int dragonCount;

    public Summoner(String name, int hitPoints, int strength, int mana, int spellPower) {
        super(name, hitPoints, strength, spellpower);
        this.summonedDragons = new Dragon [30];
        this.dragonCount = 0;
    }

    @Override
    public int castSpell(Monster target) {
         int damage = super.castSpell(target);
         return damage;
    }

    @Override
    public void summonDragon() {
        if (this.getMana() >= 2 && this.dragonCount < this.summonedDragons.length){
              this.mana -= 2;
              Dragon dragon = new Dragon("Summoned Dragon", 50, 15, 10);
              this.summonedDragons[this.dragonCount] = dragon;
              this.dragonCount++;
        }
    }

}
```