Polymorphism

Recall Inheritance

- Use the extends keyword to inherit all state and behavior from another class
- Weapon and HealthPotion both inherit "xLoc", "yLoc", "use", and the constructor from GameItem
- Weapon replaces/overrides the inherited behavior of the use method
- Super constructor must be called in subclass constructors

```java
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```java
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public int getDamage() {
        return damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

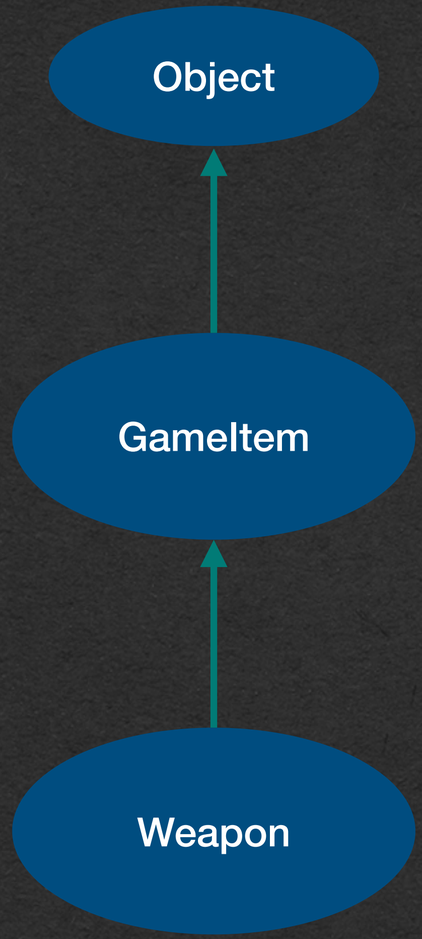
```java
public class HealthPotion extends GameItem {
    private int increase;
    public HealthPotion(double xLoc, double yLoc, int increase) {
        super(xLoc, yLoc);
        this.increase = increase;
    }
}
```

Recall Inheritance

- Weapon explicitly **extends** GameItem
- GameItem implicitly **extends** Object
- Weapon has the **state** and **behavior** of all 3 classes

```java
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```java
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public int getDamage() {
        return damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```

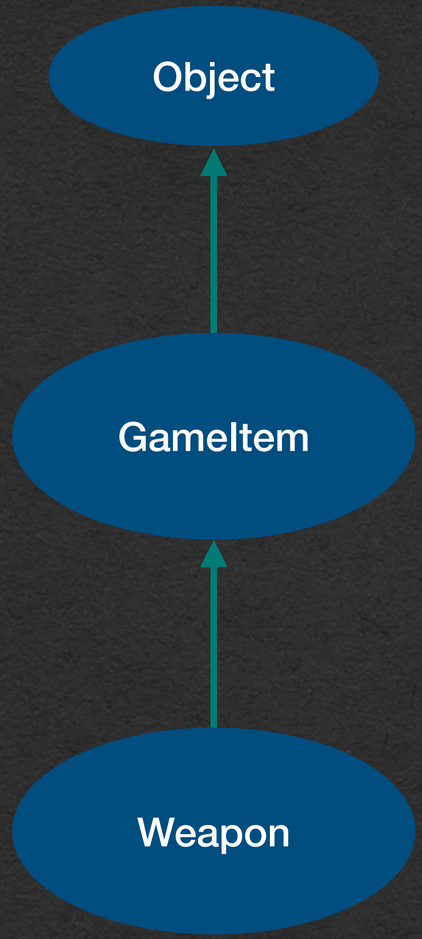


Inheritance

- When a class extends another class, we call this an "is-a" relationship
- **is-a** relationships can be direct or indirect
- Weapon **is-a** GameItem
- Weapon **is-an** Object

```java
public class GameItem {
    private double xLoc;
    private double yLoc;
    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }
    public void use() {
        System.out.println("Item Used");
    }
}
```

```java
public class Weapon extends GameItem {
    private int damage;
    public Weapon(double xloc, double yLoc, int damage) {
        super(xloc, yLoc);
        this.damage = damage;
    }
    public int getDamage() {
        return damage;
    }
    @Override
    public void use() {
        System.out.println("Damage dealt: " + this.damage);
    }
}
```



Polymorphism

If an object *is a type*

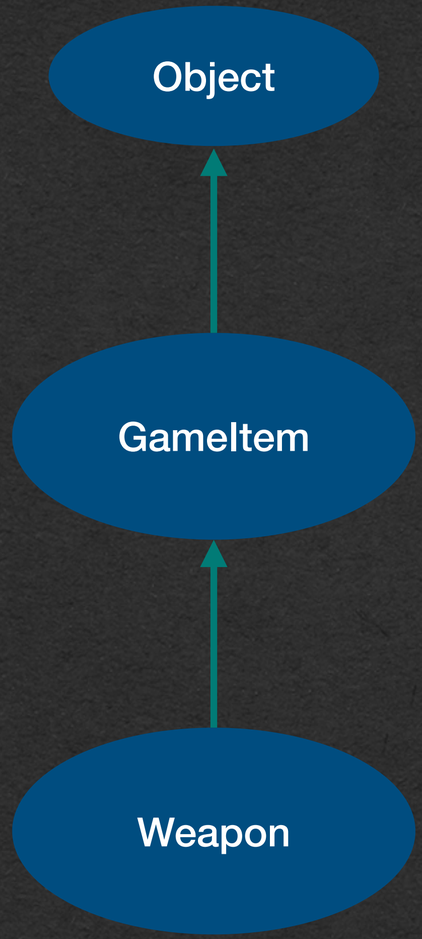
It can be stored in variables of that *type*

Polymorphism

- Weapon ***is*** 3 different types

- Polymorphism
  - Poly -> Many
  - Morph -> Forms
  - Polymorphism -> Many Forms

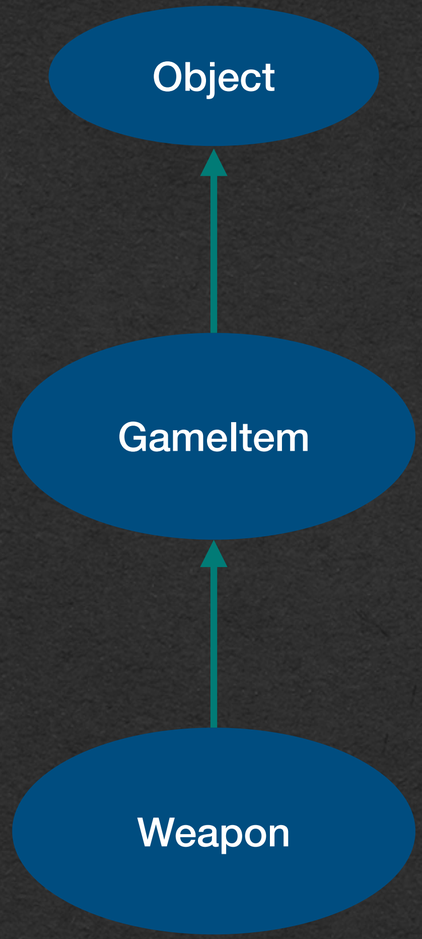
- Can store objects in variables of any of their types



Polymorphism

- All of these assignments are allowed
- Weapon has 3 different types!

```
public static void main(String[] args) {
    Weapon weapon1 = new Weapon(1.0, 1.0, 10);
    GameItem weapon2 = new Weapon(1.0, 1.0, 10);
    Object weapon3 = new Weapon(1.0, 1.0, 10);
}
```



Polymorphism

If an object *is a type*

It can be stored in variables of that *type*

Polymorphism

- Weapon has 3 different types
- Can store values in variables of any of their types

- This is polymorphism.
  - What implications does this have?

```
public static void main(String[] args) {
    Weapon weapon1 = new Weapon(1.0, 1.0, 10);
    GameItem weapon2 = new Weapon(1.0, 1.0, 10);
    Object weapon3 = new Weapon(1.0, 1.0, 10);
}
```

Polymorphism

- Can only access state and behavior of the ***variable*** type
- Defined *getDamage* in the Weapon class
- GameItem has no such method
  - Even when weapon2 stores a reference to a Weapon object, it cannot access getDamage

```
public static void main(String[] args) {
    Weapon weapon1 = new Weapon(1.0, 1.0, 10);
    GameItem weapon2 = new Weapon(1.0, 1.0, 10);
    Object weapon3 = new Weapon(1.0, 1.0, 10);
    weapon1.getDamage();
//  weapon2.getDamage(); Does not compile
//  weapon3.getDamage(); Does not compile
}
```

Polymorphism

- Can only access state and behavior of the ***variable*** type
- The use method exists in the GameItem class and is inherited by Weapon
  - Can call this method from variables of both types
- The Object class does not know about the use method
  - Cannot call use from a variable of type Object

```
public static void main(String[] args) {
    Player player = new Player(50);
    Weapon weapon1 = new Weapon(1.0, 1.0, 10);
    GameItem weapon2 = new Weapon(1.0, 1.0, 10);
    Object weapon3 = new Weapon(1.0, 1.0, 10);
    weapon1.use(player);
    weapon2.use(player);
//  weapon3.use(player); Does not compile
}
```

Polymorphism

- If the method is overridden, the override method is called *regardless* of the type of the variable
- The type of the variable determines which methods *can* be called
- The type of object determines which method *is* called

```
public static void main(String[] args) {
    Player player = new Player(50);
    Weapon weapon1 = new Weapon(1.0, 1.0, 10);
    GameItem weapon2 = new Weapon(1.0, 1.0, 10);
    Object weapon3 = new Weapon(1.0, 1.0, 10);
    weapon1.use(player);
    weapon2.use(player);
//  weapon3.use(player); Does not compile
}
```

Polymorphism

- The toString method is defined in the Object class
- Can call toString from any variable type
  - *Except primitives

```
public static void main(String[] args) {
    Player player = new Player(50);
    Weapon weapon1 = new Weapon(1.0, 1.0, 10);
    GameItem weapon2 = new Weapon(1.0, 1.0, 10);
    Object weapon3 = new Weapon(1.0, 1.0, 10);
    weapon1.toString();
    weapon2.toString();
    weapon3.toString();
}
```

Polymorphism

- Why use polymorphism if it restricts functionality?
  - Simplify other classes
- For the Player class to use a GameItem, write 2 methods
  - One to use a Weapon
  - One to use a HealthPotion
- Each item the Player can use will need another method in the Player class
- **Tedious to expand the game**

```
public class Player extends GameItem {
    private int maxHP;
    private int HP;
    private int damageDealt;

    public Player(int maxHP) {
        super(0, 0);
        this.maxHP = maxHP;
        this.HP = maxHP;
        this.damageDealt = 4;
    }

    public void useItem(GameItem item){
        item.use(this);
    }

    @Override
    void use(Player player) {
        player.setHP(player.getHP() - this.damageDealt);
    }

}
```

Polymorphism

- Instead, write a single method that takes a GameItem!
- This method can be called with a reference to a Weapon or HealthPotion as an argument
- The argument value is assigned to the parameter variable
  - This is a legal assignment because of polymorphism!
- Can add any number of GameItem classes to our game without changing the Player class
  - Easy to add more features to your game

```
public class Player extends GameItem {
    private int maxHP;
    private int HP;
    private int damageDealt;

    public Player(int maxHP) {
        super(0, 0);
        this.maxHP = maxHP;
        this.HP = maxHP;
        this.damageDealt = 4;
    }

    public void useItem(GameItem item){
        item.use(this);
    }

    @Override
    void use(Player player) {
        player.setHP(player.getHP() - this.damageDealt);
    }

}
```

Polymorphism

- In this method, we can't access any methods that are not known to the GameItem class
  - This sacrifice is often worth it for the added versatility of methods that take super types

```
public class Player extends GameItem {
    private int maxHP;
    private int HP;
    private int damageDealt;

    public Player(int maxHP) {
        super(0, 0);
        this.maxHP = maxHP;
        this.HP = maxHP;
        this.damageDealt = 4;
    }

    public void useItem(GameItem item){
        item.use(this);
    }

    @Override
    void use(Player player) {
        player.setHP(player.getHP() - this.damageDealt);
    }

}
```

Polymorphism

Polymorphism and data structures

- There's more!

- We can create data structures of a super type
- These data structures can store any type that inherits from that type

- This ArrayList of GameItems can store HealthPotions **and** Weapons!
  - We have a data structure that stores multiple different types
  - Something we took for granted in JS and Python

```
public class Player extends GameItem {
    private int maxHP;
    private int HP;
    private int damageDealt;
    private ArrayList<GameItem> inventory;

    public Player(int maxHP) {
        super(0, 0);
        this.maxHP = maxHP;
        this.HP = maxHP;
        this.damageDealt = 4;
        this.inventory = new ArrayList<>();
    }

    public void useItem(GameItem item){
        item.use(this);
    }

    public void pickUpItem(GameItem item) {
        this.inventory.add(item);
    }

    public void useAllInventoryItems() {
        for (GameItem item : this.inventory) {
            item.use(this);
        }
        this.inventory = new ArrayList<>();
    }

    @Override
    void use(Player player) {
        player.setHP(player.getHP() - this.damageDealt);
    }

}
```

Abstract

Abstract Classes

- Methods can be abstract
  - Specify the method signature (name, return type, parameters)
  - Do not define the method (no body)
  - End the method with a semicolon

```
public abstract class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }

    abstract void use(Player player);

}
```

- Abstract methods cannot be called
  - What would you expect to happen? Nothing? What if it has a return type?

Abstract Classes

- If a class has >0 abstract methods, the class itself must be abstract

- **Abstract classes cannot be instantiated**
  - Cannot create a new GameItem if GameItem is abstract
  - Prevents anyone from calling an abstract method

- They only exist to be inherited

```
public abstract class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }

    abstract void use(Player player);

}
```

Abstract Classes

- Any class inheriting from an abstract class has a requirement to implement all abstract methods
  - If the extending class overrides the abstract method, it then exists and can be called

```
public abstract class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }

    abstract void use(Player player);

}
```

- If a subclass does not implement all abstract methods, it too must be abstract

Abstract Classes

- **Why use abstract methods/classes?**

- You can only call methods that are known to your variable type
- Abstract methods are known to the abstract class
- You can call abstract methods using polymorphism

- Use an abstract method when you want all inheriting classes to have a method, but there's no clear default behavior for the method

```
public abstract class GameItem {
    private double xLoc;
    private double yLoc;

    public GameItem(double xLoc, double yLoc) {
        this.xLoc = xLoc;
        this.yLoc = yLoc;
    }

    abstract void use(Player player);

}
```

Interfaces

- If we take this one step further, we can create **interfaces**
- Interfaces are similar to classes
- Interfaces can **only** have abstract methods
  - No instance variables
  - No constructor
  - No methods with definitions
- To inherit an interface, use the implements keyword instead of extends

```
public interface Comparator<T> {
   boolean compare(T a, T b);
}
```

```
public class IntDecreasing implements Comparator<Integer> {
    @Override
    public boolean compare(Integer a, Integer b) {
        return a > b;
    }
}
```

Interfaces

- Why interfaces?
- You can only extend one class
- You can implement as many interfaces as you'd like
- *This avoids the potential of multiple definitions for the same method

```
public interface Comparator<T> {
   boolean compare(T a, T b);
}
```

```
public class IntDecreasing implements Comparator<Integer> {
    @Override
    public boolean compare(Integer a, Integer b) {
        return a > b;
    }
}
```