Object-Oriented Programming

Class Relationships

Association, Aggregation, Composition

Outline

- Recall (Exercise 4)
- Overview of Class Relationships
- Association
- Aggregation
- Composition

Before we begin

Any questions with the Exercise 4? ☺️

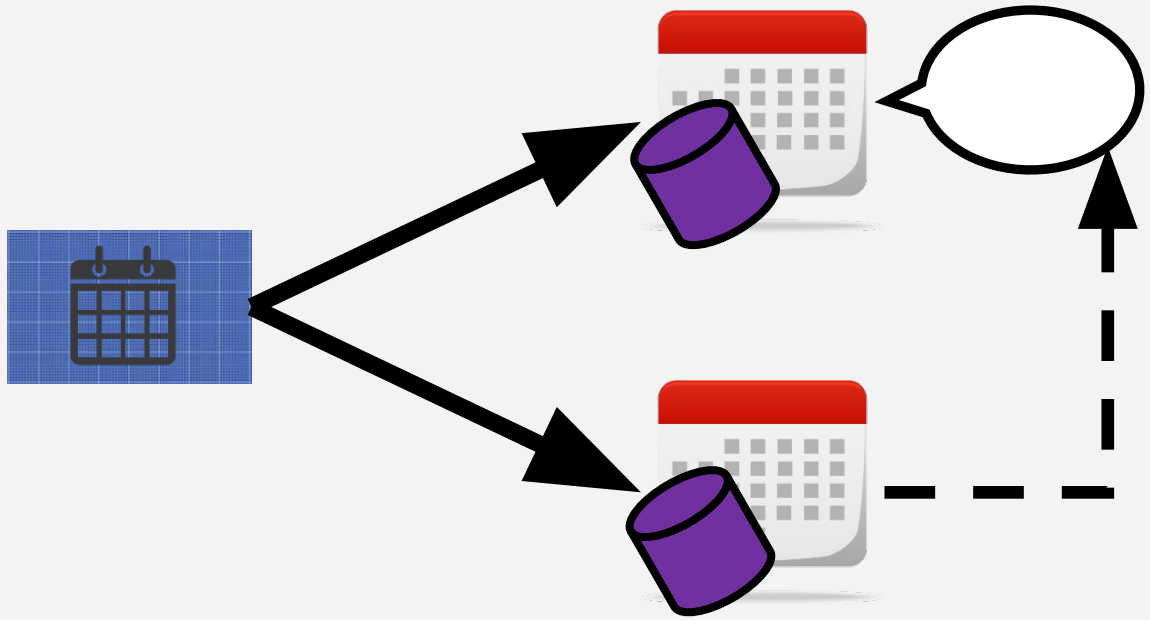
Recall: MyDate Exercise

MyDate
- day : int - month : int - year : int
+ Date() + Date(month : int, day : int) + Date(year : int, month : int, day : int) **+ isBefore(date : MyDate) : boolean** + setYear(year : int) : void + setMonth(month : int) : void + setMonth(month : String) : void + setDay(day : int) : void + getYear () : int + getMonth() : int + getDay() : int

We haven't explicitly dealt with reference types as method parameters – particularly those that we've created – so this may have seemed strange to some

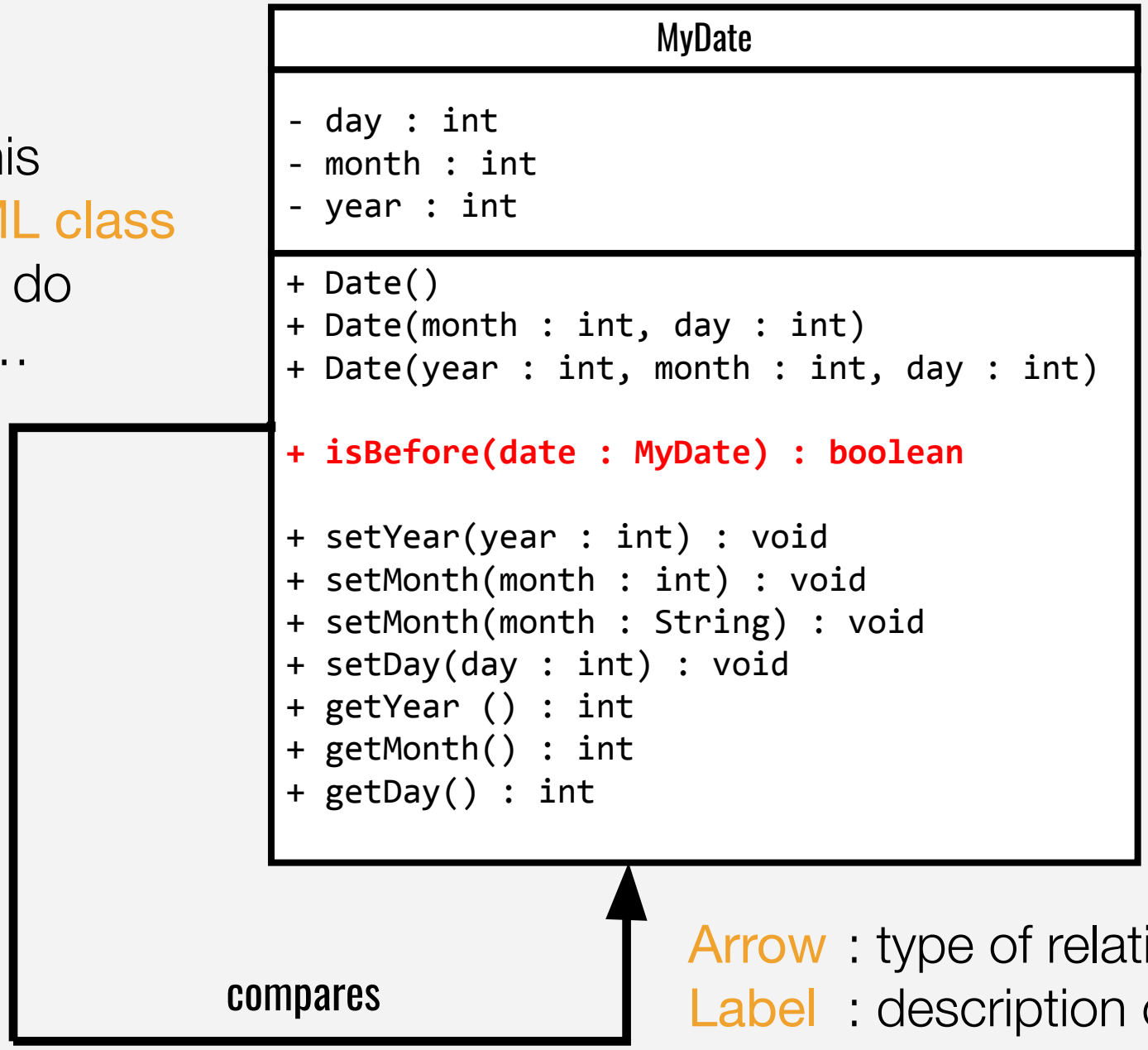
Recall: MyDate Exercise

```
public boolean isBefore(MyDate date) {
    // returns true if this date comes
    // before the supplied date, and
    // false otherwise.
}
```



Observing this method, we can say that MyDate objects interact with other MyDate objects – that there is some kind of relationship between MyDate objects

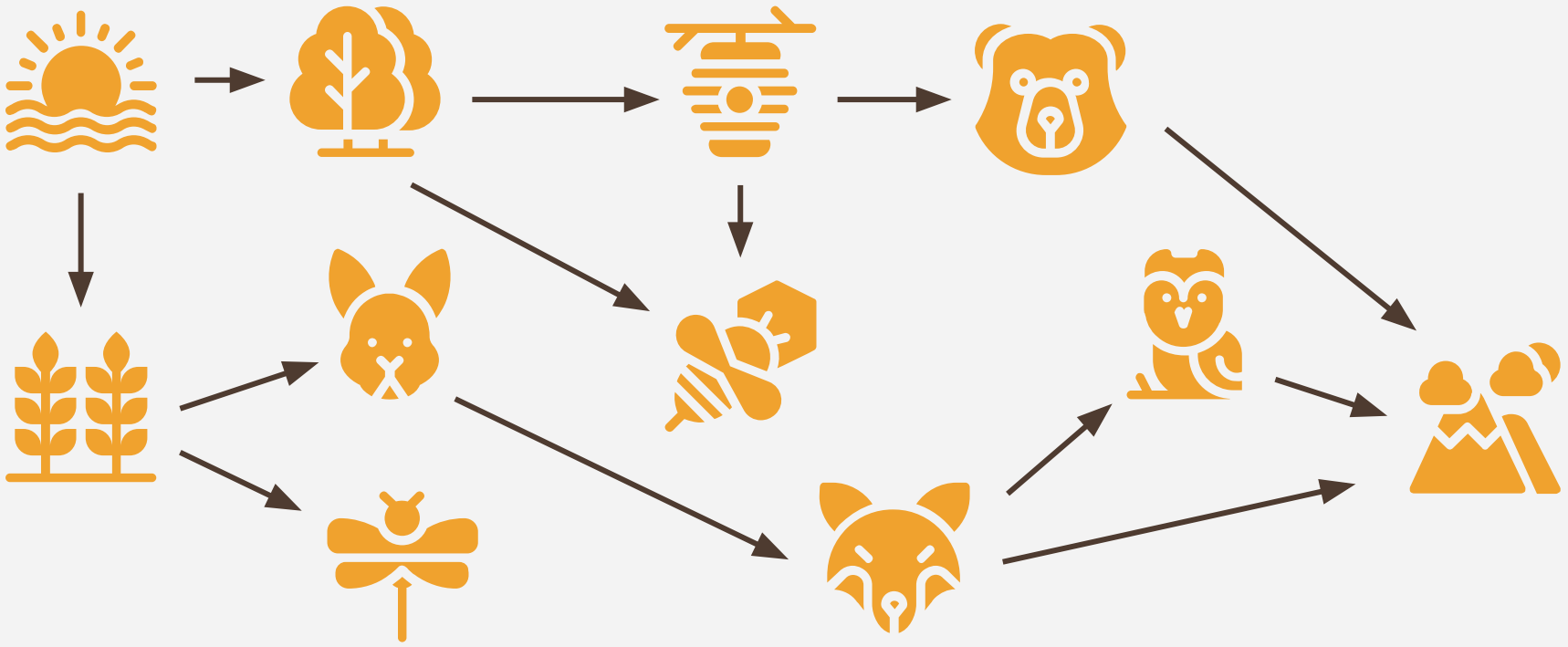
And to represent this relationship in a UML class diagram, we would do something like this…



Arrow : type of relationship
Label : description of relationship

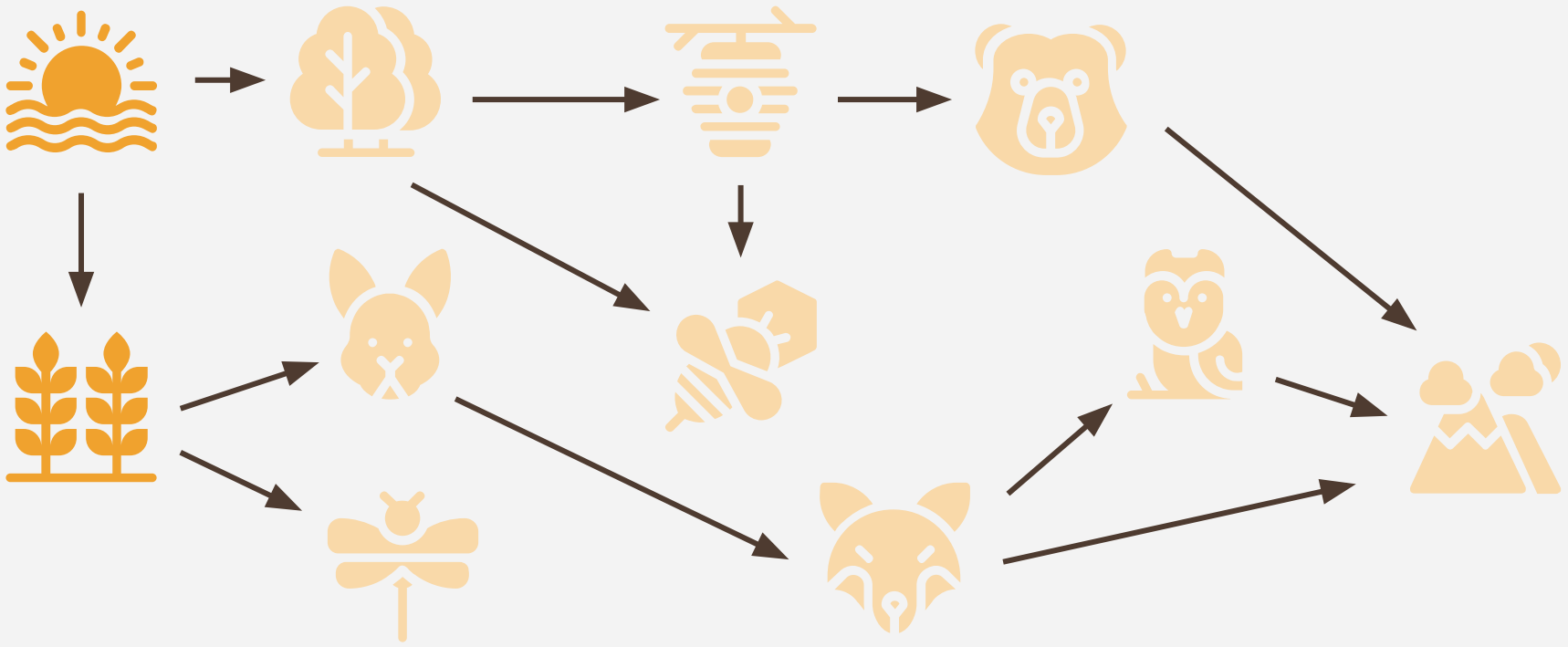
Class Relationships

Remember this example during our introduction…



Class Relationships

… and how we focused on the Sun affecting the Plant's growth?



Recall: object-oriented solutions solve problems by having independent entities (objects) interact with each other, with each entity performing their own roles in a well-crafted system

Class Relationships

- Objects cannot perform much by themselves
  - Otherwise, they become god-objects which violate OOP
- Objects should interact with other objects in order to accomplish complex tasks

- Hence, Class Relationships defines how classes relate or interact with each other

Class Relationships

- Class relationships fall under four major types

We'll tackle Inheritance later in the term ☺️

Throughout this lecture, we will look at how class relationships are denoted in UML and implemented in code. However, keep in mind that class relationships are design decisions. There may be multiple ways to implement a certain relationship, so example implementations simply serve as guides.

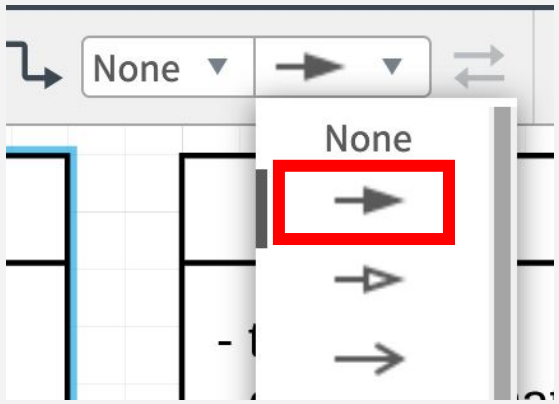
Association

- Describes a “using” relationship
  - Player uses/drinks HealingPotion
  - MyDate uses/checks MyDate

- If a class uses/receives a class in any of its methods, then there is an association relationship between those two classes

- Objects in this relationship maintain their own life cycle
  - Meaning, the destruction of one does not cause the destruction of another

Association

- UML: Use the 

arrow

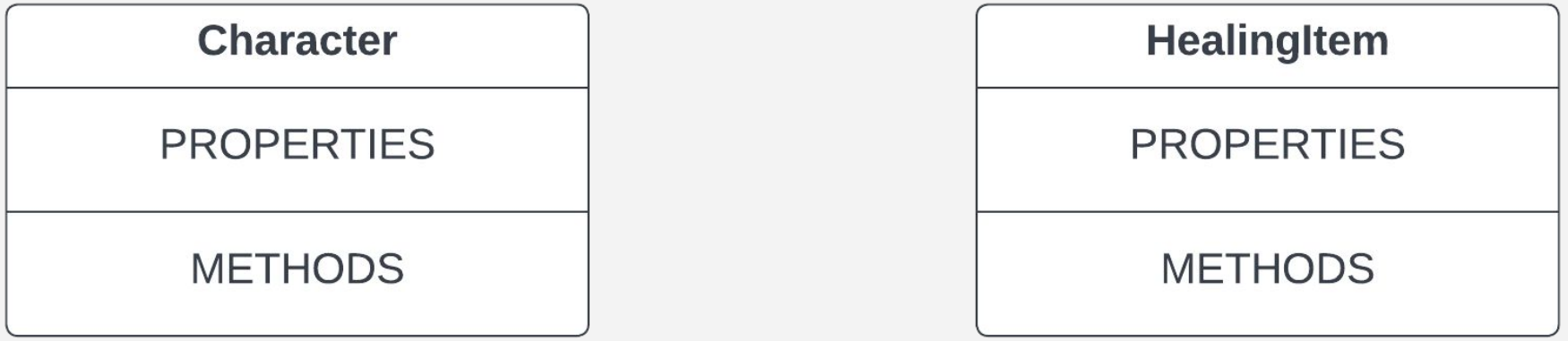
- Furthermore, there are three types of Association relationships
  1. Directed
  2. Bidirectional
  3. Reflexive

Association – Directed

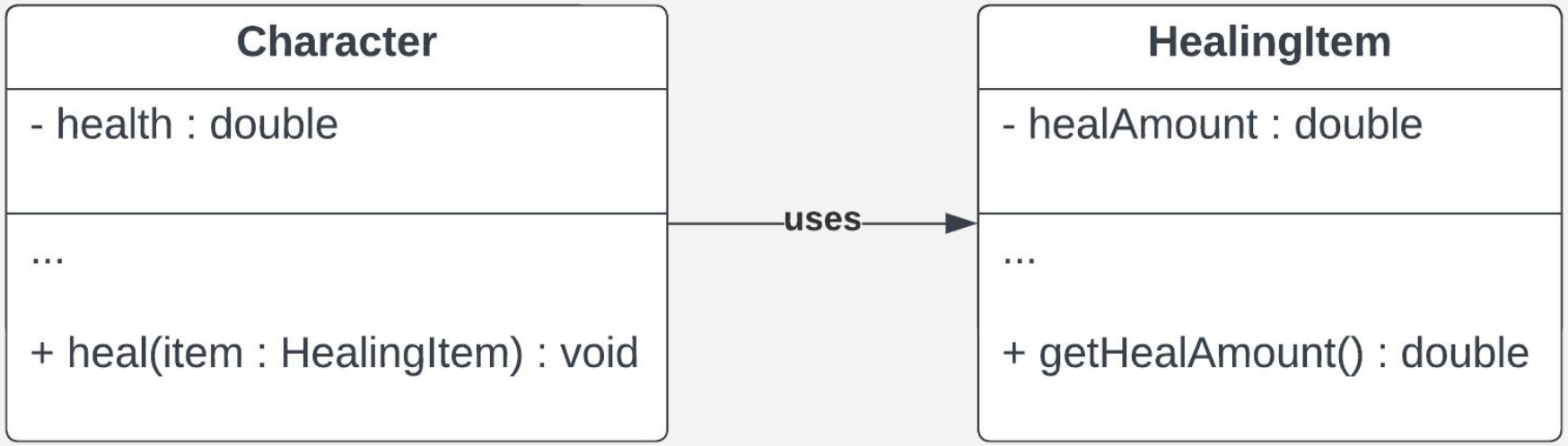
- A one-way association
  - Only instances of one class use instances of the other class
- Classes whose instances share a directed association relationship with another object would often have a method that accepts instances of the other class as its parameter

Association – Directed

Let’s say a Character uses a HealingItem to increase their health…

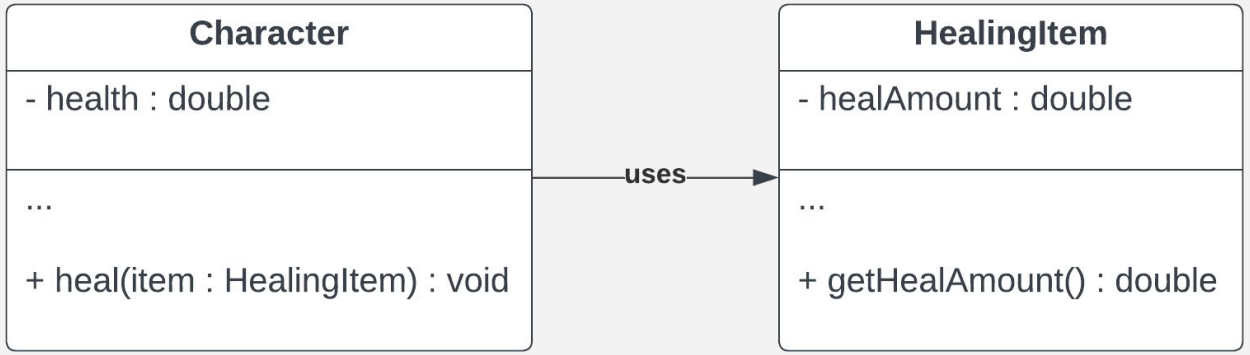


Association – Directed



Association – Directed

The accepting object does not directly modify the properties of the parameter object (remember encapsulation!)




```
public class Character {
   private double health;

   // Other parts of the class here

   public void heal(HealingItem item) {
      this.health = this.health + item.getHealAmount();
   }
}
```

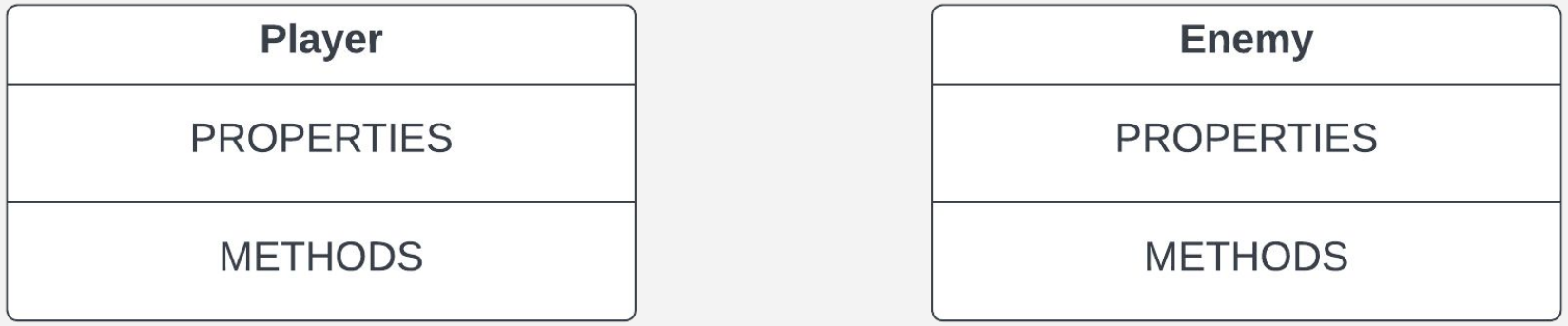
```
public class HealingItem {
   private double healAmount;

   // Other parts of the class here

   public double getHealAmount() {
      return this.healAmount;
   }
}
```

Association – Bidirectional

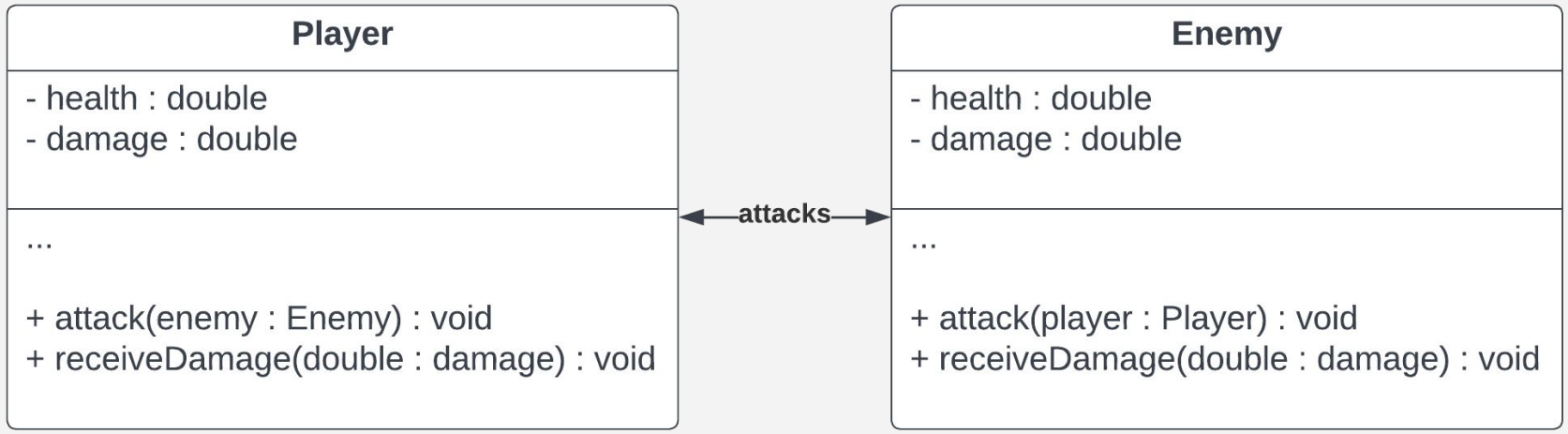
- A two-way association
  - Instances of both classes can perform the associative action to each other
  - These classes must possess methods that enable the performance of their actions

Association – Bidirectional

Let’s say a Player and an Enemy can attack/damage each other…

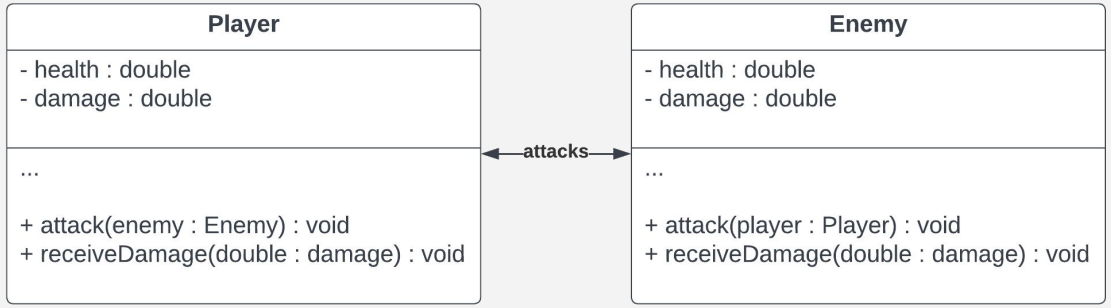


Association – Bidirectional



Since the attack() method is the same for both classes, we can use 1 arrow with both arrow heads. If the actions/methods are different, it would be better to use separate arrows

Association – Bidirectional




```
public class Player {
   private double health;
   private double damage;

   // Other parts of the class here

   public void attack(Enemy enemy) {
      enemy.receiveDamage(this.damage);
   }

   public void receiveDamage(double damage) {
      this.health = this.health – damage;
   }
}
```

```
public class Enemy {
   private double health;
   private double damage;

   // Other parts of the class here

   public void attack(Player player) {
      player.receiveDamage(this.damage);
   }

   public void receiveDamage(double damage) {
      this.health = this.health – damage;
   }
}
```

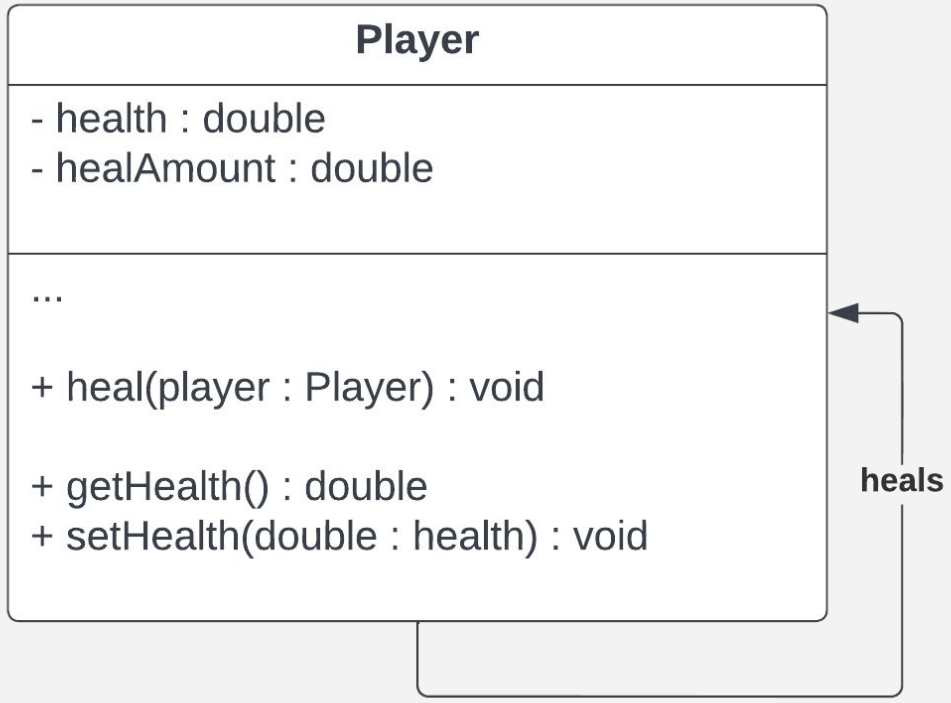
Association – Reflexive

- Exists between separate instances of the same class OR to the same instance of a class itself
  - A class must have a method that accepts instances of itself as a parameter
  - Another variant would be to perform the action on itself directly

Association – Reflexive

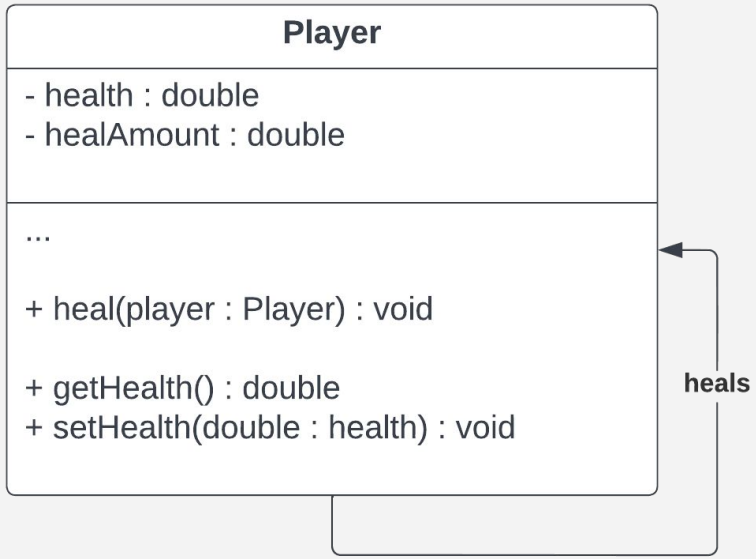
Let’s say a Player can heal other players… including itself…

Association – Reflexive



Association – Reflexive

```
public class Player {
   private double health;
   private double healAmount;

   // Other parts of the class here

   public void heal(Player player) {
      player.setHealth(player.getHealth() + this.healAmount);
   }

   public double getHealth() {
      return this.health;
   }

   public void setHealth(double health) {
      this.health = health;
   }
}
```

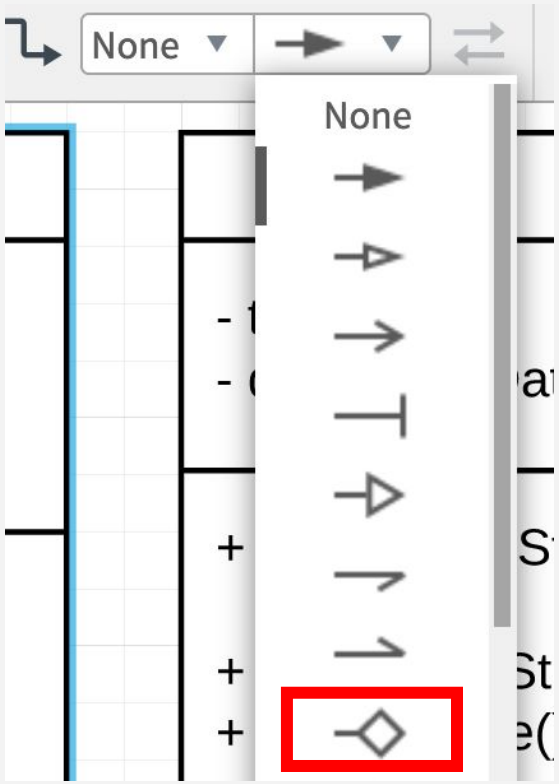


Questions? ☺️

Aggregation

- A special, more specific kind of Association
- Describes a part-whole relationship
  - Whole “has a” part | Part “of” a whole relationship
- Like association, the objects still have their own life cycles
- Unlike association, aggregation demands that one be the “whole” class and the other be the “part”

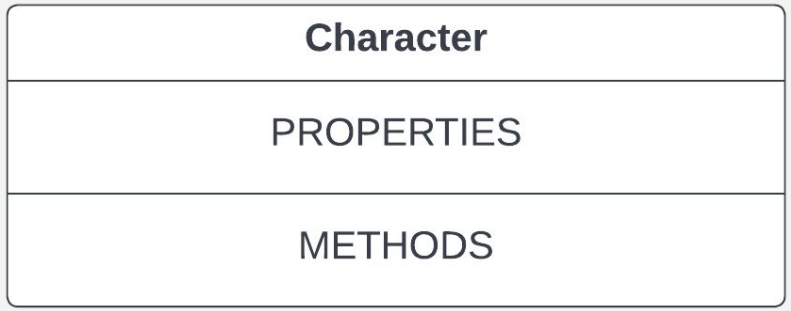
Aggregation

- UML: Use the 

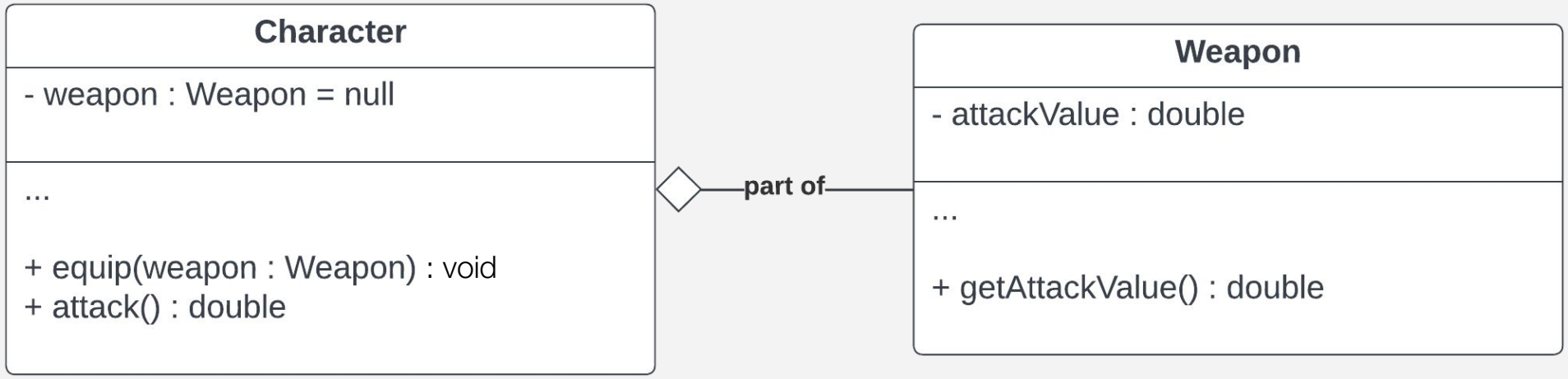
arrow

Aggregation

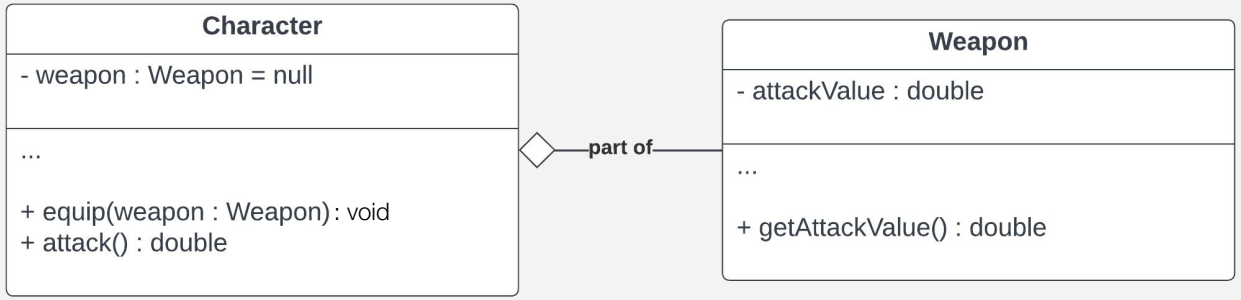
Let’s say a Character equips a Weapon, which affects one’s damage…



Aggregation



Aggregation




```
public class Character {
   private Weapon weapon = null;

   // Other parts of the class here

   public void equip(Weapon weapon) {
      this.weapon = weapon;
   }

   public double attack() {
      double damage = 0;
      if(this.weapon != null) {
         damage = this.weapon.getAttackValue();
      }
      return damage;
   }
}
```

The description can be written from the “part” perspective *(e.g part of)* or from the “whole” perspective *(e.g. equips)*.

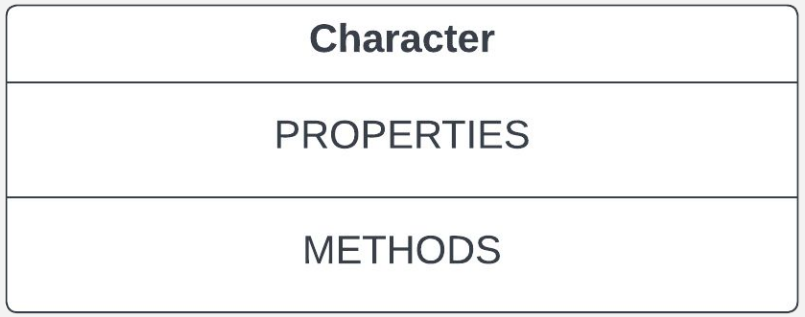
```
public class Weapon {
   private double attackValue;

   // Other parts of the class here

   public double getAttackValue() {
      return this.attackValue;
   }
}
```

Composition

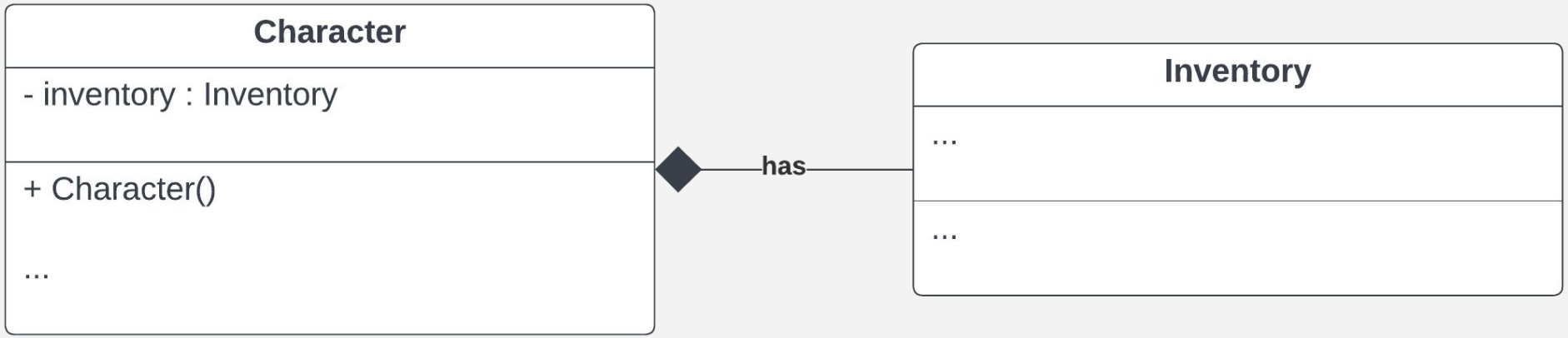
- A special, more specific kind of Aggregation
  - Like aggregation, the classes in this relationship exhibit a part-whole relationship (specifically “part-of”)
  - However, one class is dependent on the existence of the other class
    - Destroying the “whole” or “owner” also destroys the “part”
    - The “part” instance is usually created by the “whole”

Composition

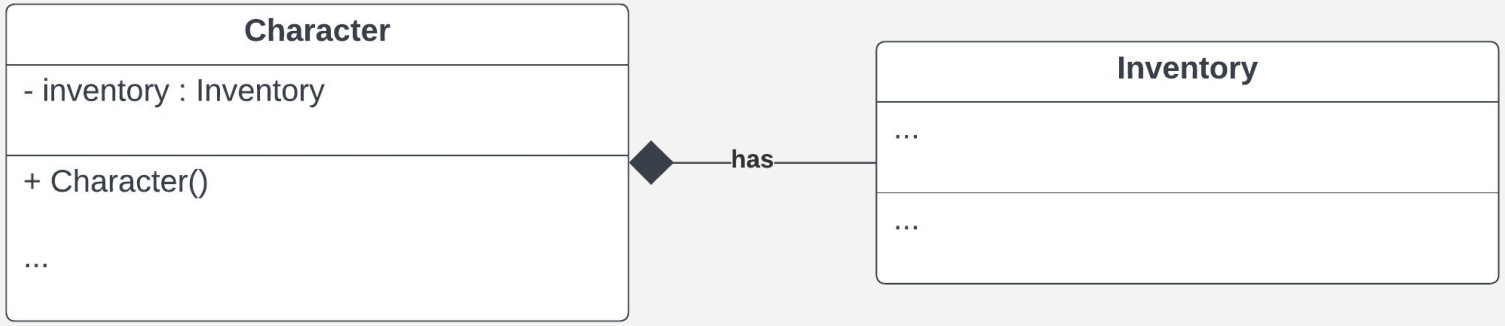
Let’s say a Character has an Inventory…



Composition



Composition




```
public class Character {
   private Inventory inventory;

   public Character() {
      this.inventory = new Inventory();
   }

   // Other parts of the class here
}
```

```
public class Inventory {
   // Other parts of the class here
}
```

In this example, an Inventory object is centered in the Character class. Assuming the Inventory object can't be passed out, destroying a Character object would result in the destruction of the Inventory instance.

Object Life Cycle Disclaimer

- Don’t take real-world analogies too literally…
  - If a house “has a” tenant and the house was destroyed with the tenant “part of” the house, how does the tenant survive?
    - Not a good way to look at this!
  - When reference types are passed by through parameters, the reference/address is being passed
- Remember, we’re looking at design considerations for eventual implementation of some system

Questions? 

Keep learning…