Polymorphism

Lecture Task

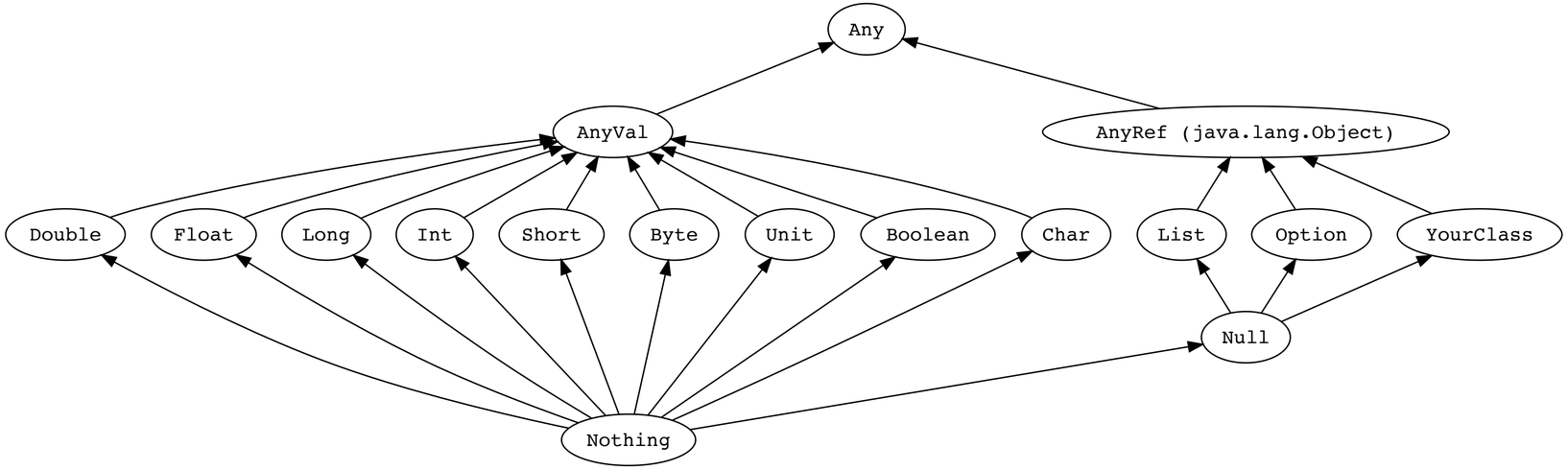
- Point of Sale: Lecture Task 4 -

Functionality: Update your Item class so Items can have Modifiers applied to them by adding:

- A method named “addModifier” that takes a Modifier as a parameter and returns Unit
  - After a modifier is added with this method, that modifier should be applied to all future method calls
- Update the “price” method to apply all modifiers to the price of the item
  - Tax must not be included in this price
- A method named “tax” that takes no parameters and returns the total tax applied to this item from all of its modifiers

Testing: In the tests package, create a test suite named LectureTask4 that tests this functionality.

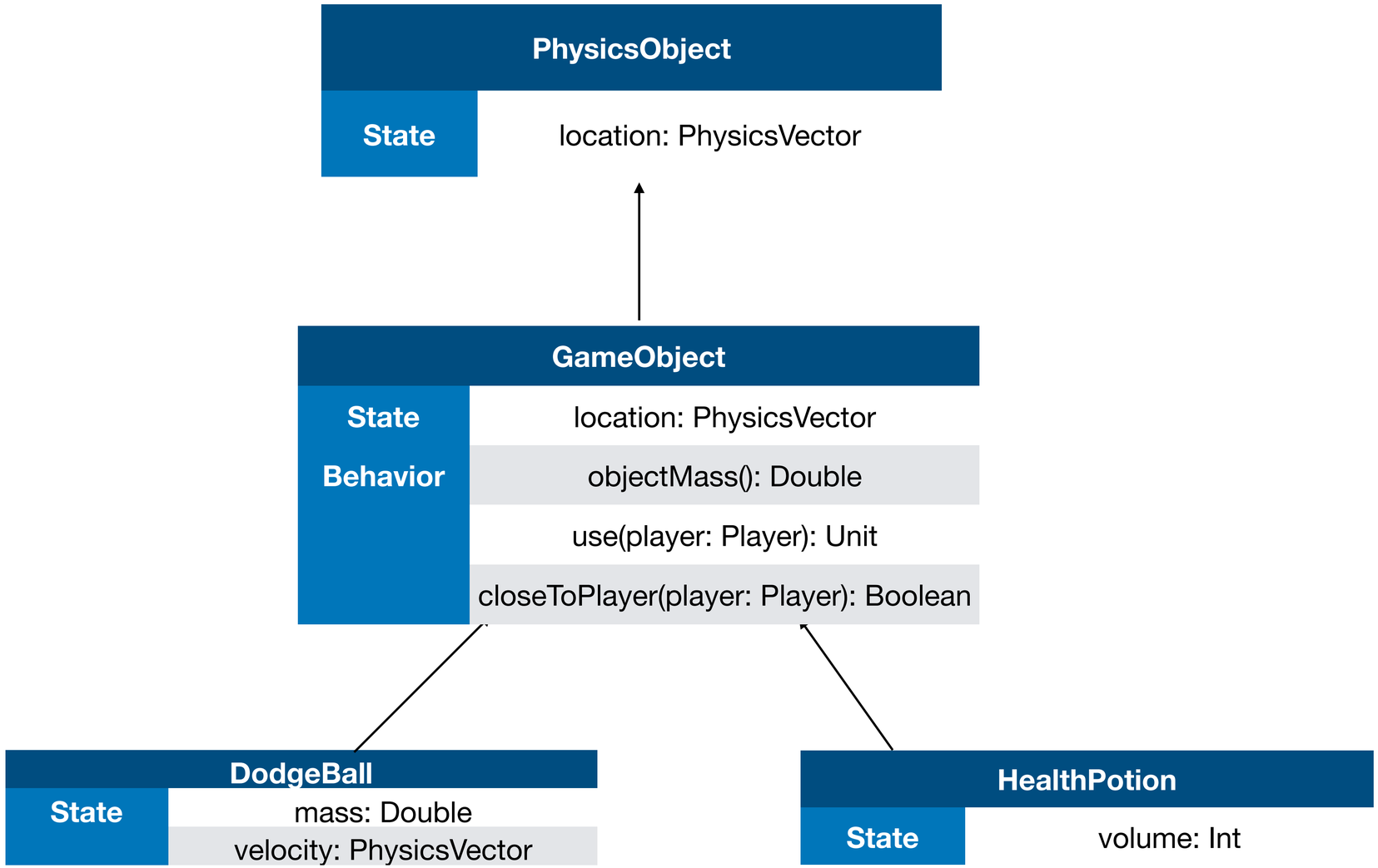
Scala Type Hierarchy



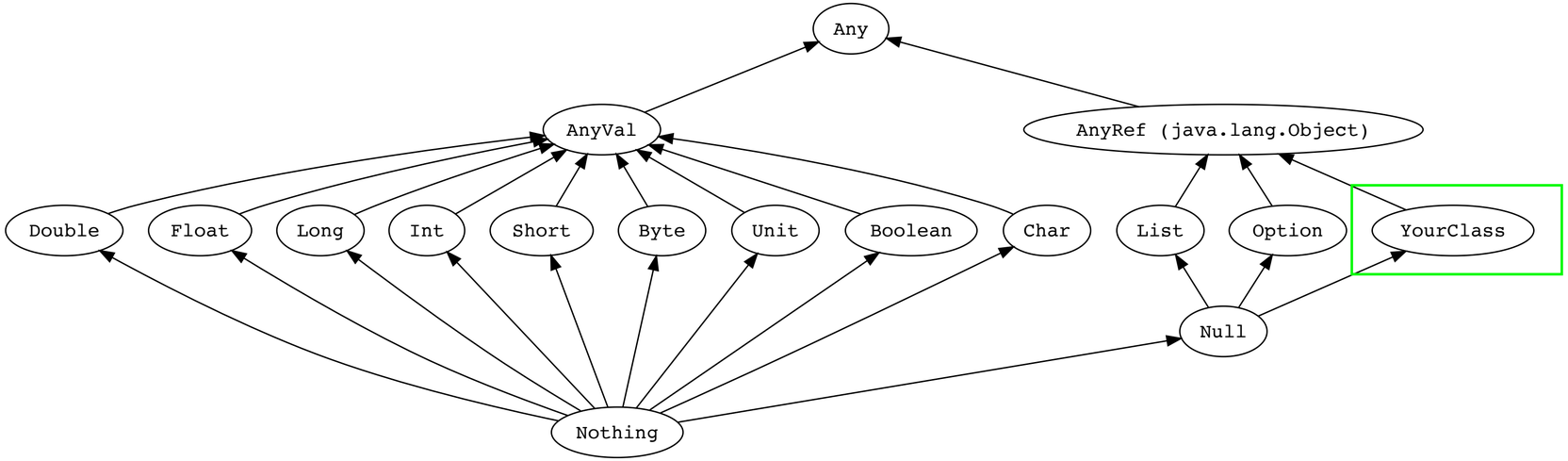
- All objects share Any as their base types
- Classes extending AnyVal will be stored on the **stack**
  - *Unless they are a state variable of an object
- Classes extending AnyRef will be stored on the **heap**

https://docs.scala-lang.org/tour/unified-types.html

Recall


PhysicsObject
State
location: PhysicsVector
GameObject
State
location: PhysicsVector
Behavior
objectMass(): Double
use(player: Player): Unit
closeToPlayer(player: Player): Boolean
DodgeBall
State
mass: Double
velocity: PhysicsVector
HealthPotion
State
volume: Int

Scala Type Hierarchy



- Classes you define extend AnyRef by default
- HealthPotion has 5 different types

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion2: GameObject = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion3: PhysicsObject = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion4: AnyRef = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion5: Any = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
```

https://docs.scala-lang.org/tour/unified-types.html

Polymorphism

- HealthPotion has 5 different types
- Polymorphism
  - Poly -> Many
  - Morph -> Forms
  - Polymorphism -> Many Forms
- Can store values in variables of any of their types

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion2: GameObject = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion3: PhysicsObject = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion4: AnyRef = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion5: Any = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
```

Polymorphism

- Can only access state and behavior defined in variable type
- Defined distanceToPlayer in GameObject
- HealthPotion inherited distanceToPlayer when it extended GameObject
- PhysicsObject has no such method
  - Even when potion3 stores a reference to a HealthPotion object, it cannot access distanceToPlayer

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion2: GameObject = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion3: PhysicsObject = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion4: AnyRef = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion5: Any = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)

potion1.distanceToPlayer(player)
potion2.distanceToPlayer(player)
potion3.distanceToPlayer(player) // Does not compile
```

Polymorphism

- Why use polymorphism if it restricts functionality?
  - Simplify other classes
- Player has 2 methods
  - One to use a ball
  - One to use a potion
- Each item the Player can use will need another method in the Player class
- **Tedious to expand game**

```
class Player(var location: PhysicsVector,
             var orientation: PhysicsVector,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

  def useBall(ball: Ball): Unit = {
    ball.use(this)
  }

  def useHealthPotion(potion: HealthPotion): Unit = {
    potion.use(this)
  }
}
```

Polymorphism

- Write functionality using the common base type
- The use method is part of GameObject
- Can't access any Ball or HeathPotion specific functionality
  - Any state/behavior needed by Player must be in the InanimateObject class

```
abstract class GameObject(
          location: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

}
```

```
class Player(var location: PhysicsVector,
             var orientation: PhysicsVector,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

  def useBall(ball: Ball): Unit = {
    ball.use(this)
  }

  def useHealthPotion(potion: HealthPotion): Unit = {
    potion.use(this)
  }
}
```

```
class Player(var location: PhysicsVector,
             var orientation: PhysicsVector,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

  def useItem(item: GameObject): Unit = {
    item.use(this)
  }

}
```

Polymorphism

- We can call useItem with any object that extends InanimateObject as an argument
- The useItem method will have different effects depending on the type of its parameter
  - Different implementations of use will be called
- Adding new object types to our game does not require changing the Player class!
  - Test Player once
  - Without polymorphism we'd have to update and test the Player class for every new object type added to the game

```
abstract class GameObject(
          location: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

}
```

```
class Player(var location: PhysicsVector,
             var orientation: PhysicsVector,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

  def useItem(item: GameObject): Unit = {
    item.use(this)
  }

}
```

```
val ball: Ball = new Ball(new PhysicsVector(), 5)
val potion: HealthPotion = new HealthPotion(new PhysicsVector(), 5)

val player1: Player = new Player(new PhysicsVector(), new PhysicsVector(), 20, 12)

player1.useItem(ball)
player1.useItem(potion)
```

Polymorphism

- We can also make our player be a PhysicsObject

```
class Player(var location: PhysicsVector,
             var orientation: PhysicsVector,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

  def useItem(item: InanimateObject): Unit = {
    item.use(this)
  }

}
```



```
class Player(_location: PhysicsVector,
             var orientation: PhysicsVector,
             val maxHealth: Int,
             val strength: Int) extends GameObject(_location) {

  var health: Int = maxHealth

  def useItem(item: GameObject): Unit = {
    item.use(this)
  }

}
```

Polymorphism

- With polymorphism, we can mix types in data structures
  - Something we took for granted in Python/JavaScript
- Assume we have a physics engine that takes a List of PhysicsObjects
- If all our objects are PhysicsObjects, put them in a list and send them to the physics engine

```
val player: Player = new Player(new PhysicsVector(0.0, 0.0),
  new PhysicsVector(1.0, 0.0), 10, 255)

val potion1: HealthPotion = new HealthPotion(new PhysicsVector(-8.27, -3.58), 6)
val potion2: HealthPotion = new HealthPotion(new PhysicsVector(-8.046, -2.128), 6)
val ball: DodgeBall = new DodgeBall(new PhysicsVector(-2.28, 4.88, 5.1689),
  new PhysicsVector(1.0, 1.0, 1.0), 2)

val gameObjects: List[PhysicsObject] = List(player, potion1, potion2, ball)

PhysicsEngine.doPhysics(gameObjects)
```

Polymorphism

- HealthPotion has 5 different types
- Can store values in variables of any of their types

- This is polymorphism
  - What implications does this have?

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion2: GameObject = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion3: PhysicsObject = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion4: AnyRef = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
val potion5: Any = new HealthPotion(new PhysicsVector(0.0, 0.0), 6)
```

Override

Override

- Functionality is inherited from Any and AnyRef
- println calls an inherited .toString method
  - Converts object to a String with <object_type>@<reference>
- == calls the inherited .equals method
  - returns true only if the two variables refer to the same object in memory

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(0,0), 4)
val potion2: HealthPotion = new HealthPotion(new PhysicsVector(0,0), 4)
val potion3 = potion1

println(potion1)
println(potion2)
println(potion3)
println(potion1 == potion2)
println(potion1 == potion3)
```

```
lo2_oop.oop_physics.with_oop.HealthPotion@17c68925
lo2_oop.oop_physics.with_oop.HealthPotion@7e0ea639
lo2_oop.oop_physics.with_oop.HealthPotion@17c68925
false
true
```

Override

- We can override this default functionality
- Override toString to return a different string

```
class HealthPotion(location: PhysicsVector,val volume: Int)
  extends GameObject(location) {

...

  override def toString: String = {
    "location: " + this.location + "; volume: " + volume
  }

}
```

```
class PhysicsVector(var x: Double, var y: Double, var z: Double) {

  override def toString: String = {
    "(" + x + ", " + y + ", " + z + ")"
  }

}
```

Override

- Override equals to change the definition of equality
- Takes Any as a parameter
- Use match and case to behave differently on different types
- The _ wildcard covers all types not explicitly mentioned
- This method return true when compared to another potion with the same volume, false otherwise

```
class HealthPotion(location: PhysicsVector, val volume: Int)
  extends GameObject(location) {

...

  override def equals(obj: Any): Boolean = {
    obj match {
      case hp: HealthPotion => this.volume == hp.volume
      case _ => false
    }
  }

}
```

Override

- With our overridden methods this code gives very different output

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(0,0), 4)
val potion2: HealthPotion = new HealthPotion(new PhysicsVector(0,0), 4)
val potion3 = potion1

println(potion1)
println(potion2)
println(potion3)
println(potion1 == potion2)
println(potion1 == potion3)
```

```
location: (0.0, 0.0); volume: 4
location: (0.0, 0.0); volume: 4
location: (0.0, 0.0); volume: 4
true
true
```

Lecture Task

- Point of Sale: Lecture Task 4 -

Functionality: Update your Item class so Items can have Modifiers applied to them by adding:

- A method named “addModifier” that takes a Modifier as a parameter and returns Unit
  - After a modifier is added with this method, that modifier should be applied to all future method calls
- Update the “price” method to apply all modifiers to the price of the item
  - Tax must not be included in this price
- A method named “tax” that takes no parameters and returns the total tax applied to this item from all of its modifiers

Testing: In the tests package, create a test suite named LectureTask4 that tests this functionality.