Physics

- Acceleration
  - new velocity = old velocity + acceleration * delta time
  - $v = v_0 + a * dt$
- Distance
  - new distance = old distance + velocity * delta time
  - $d = d_0 + v * dt$
- Each dimension (x, y, z) is computed individually
  - Acceleration only in the z direction

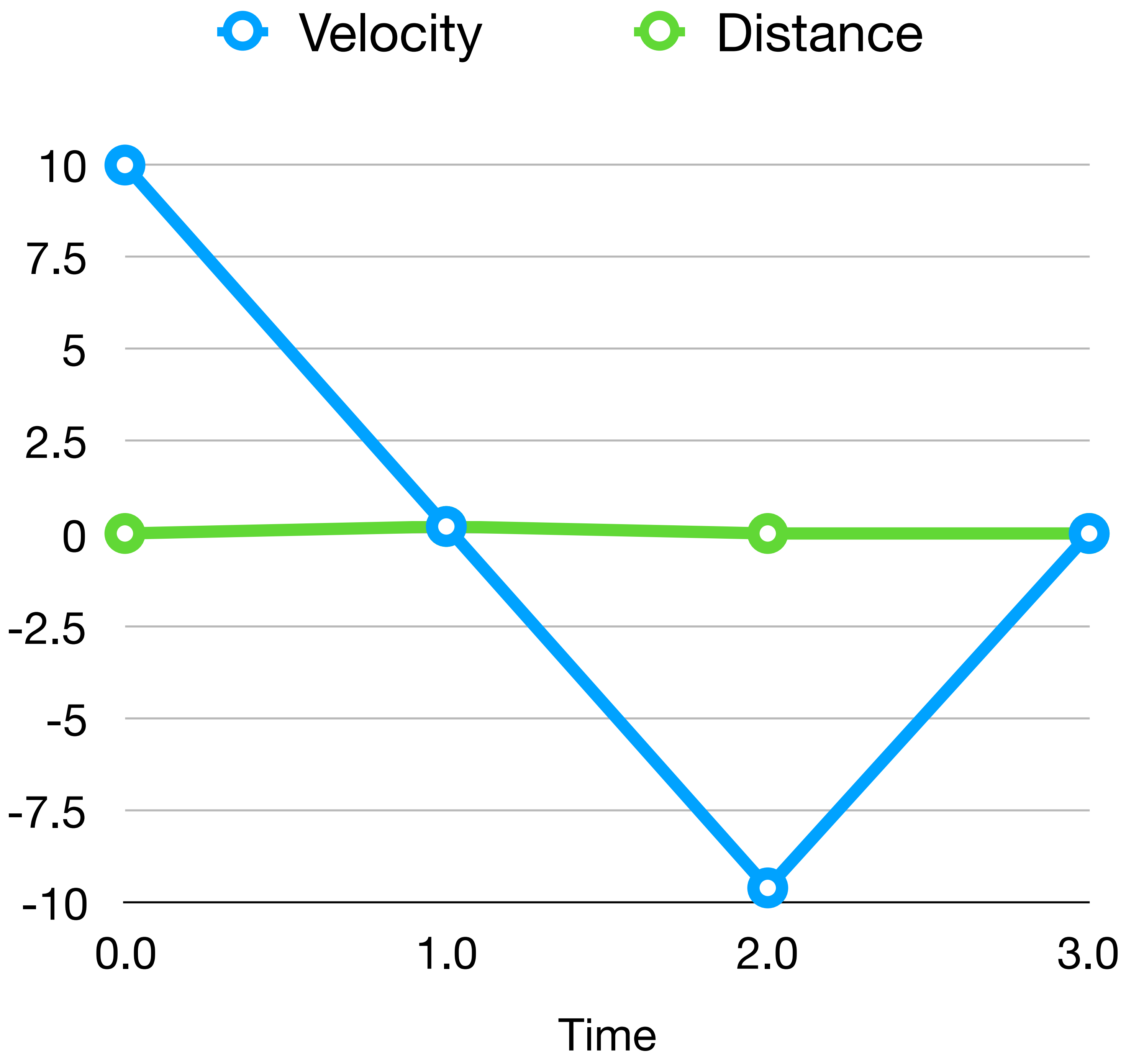
Physics Application

```scala
def main(args: Array[String]): Unit = {
  // Create a new simulation of Earth with metric units
  val earth: World = new World(9.81)

  // Add a ball to the world that is throw straight up at a velocity of 10 m/s
  val ball: PhysicalObject = new PhysicalObject(new PhysicsVector(0.0, 0.0, 0.0), new PhysicsVector(0.0, 0.0, 10.0))
  earth.objects = List(ball)


  var time: Double = 0.0
  var endOfTime: Double = 2.5

  var deltaTime: Double = 1.0

  var times = List(time)
  var zVelocity = List(ball.velocity.z)
  var height = List(ball.location.z)

  // Simulate the physics of Earth
  while(time < endOfTime){
    Physics.updateWorld(earth, deltaTime)
    time += deltaTime

    times = times :+ time
    zVelocity = zVelocity :+ ball.velocity.z
    height = height :+ ball.location.z
  }
  println(times.mkString("\t"))
  println(zVelocity.mkString("\t"))
  println(height.mkString("\t"))
}
```

Physics Application

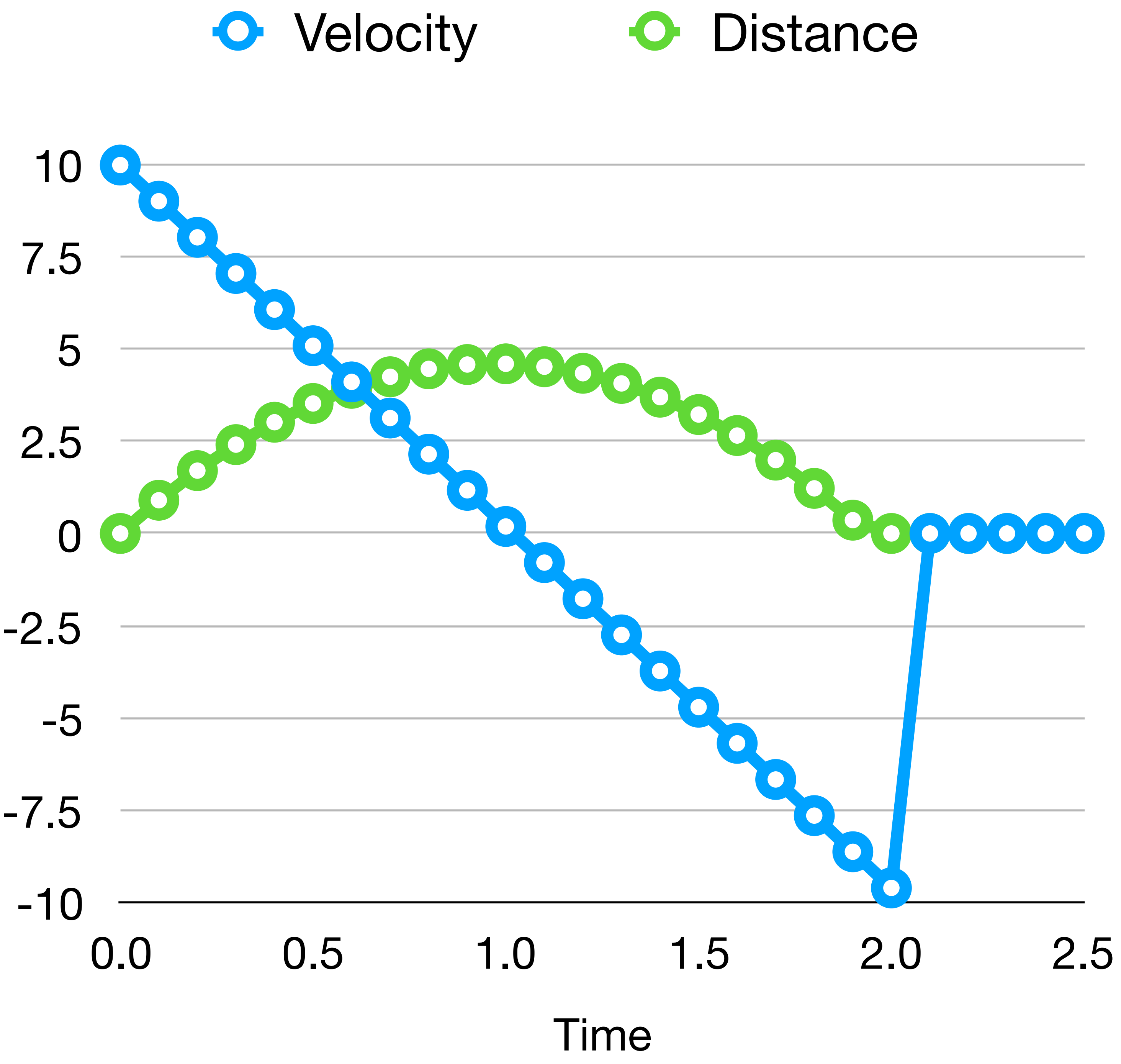
- $v = v_0 + a * dt$
- $d = d_0 + v * dt$
- $v_{z0} = 10$
- $g = 9.81$
- $dt = 1.0$

- That's not good



Physics Application

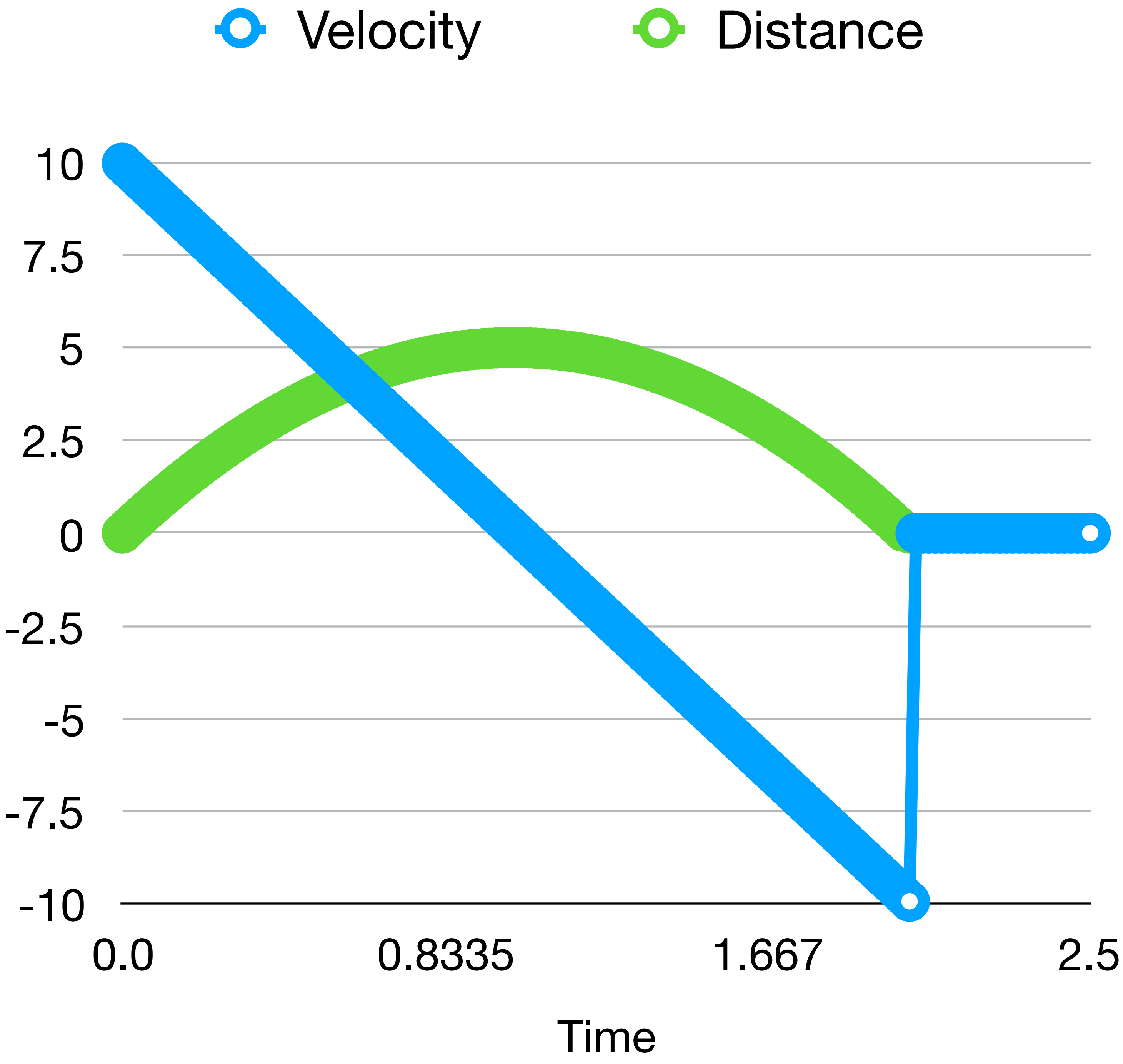
- $v = v_0 + a * dt$
- $d = d_0 + v * dt$
- $v_{z0} = 10$
- $g = 9.81$
- **dt = 0.1**



- As dt approaches 0.0
    - Simulation becomes more accurate

Physics Application

- $v = v_0 + a * dt$
- $d = d_0 + v * dt$
- $v_{z0} = 10$
- $g = 9.81$
- **dt = 0.01667**
- **And many games run at 60 FPS**
  - Mostly accurate physics without computing integrals



Inheritance

Overview

- Let's do some world building
- If we're making a game (we're making a game) we'll want various objects that will interact with each other
- We'll setup a simple where
    - Each player has a set health and strength
    - Players can pick up and throw balls
    - If a player gets hit with a ball, they lose health
    - Players can collect health potions to regain health

Objects Review

- We'll need different objects for this game
  - Player
  - Ball
  - HealthPotion

Objects Review

Player		
State	location: PhysicsVector	(2.0, -2.0, 2.0)
	velocity: PhysicsVector	(0.0, -1.0, 0.0)
	orientation: PhysicsVector	(0.5, -0.5, 0.0)
	health: Int	17
	maxHealth: Int	20
	strength: Int	25
Behavior	useBall(ball: Ball): Unit	
	useHealthPotion(potion: HealthPotion): Unit	

```
object Player {
  var location: PhysicsVector = new PhysicsVector(2.0, -2.0, 2.0)
  var velocity: PhysicsVector = new PhysicsVector(0.0, -1.0, 0.0)
  var orientation: PhysicsVector = new PhysicsVector(0.5, -0.5, 0.0)

  val maxHealth: Int = 20
  val strength: Int = 25

  var health: Int = 17

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

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

Objects Review

Ball		
State	location: PhysicsVector	(1.0, 5.0, 2.0)
	velocity: PhysicsVector	(1.0, 1.0, 10.0)
	mass: Double	5.0
Behavior	used(player: Player): Unit	

```scala
object Ball {
  var location: PhysicsVector = new PhysicsVector(1.0, 5.0, 2.0)
  var velocity: PhysicsVector = new PhysicsVector(1.0, 1.0, 10.0)
  val mass: Double = 5.0

  def use(player: Player): Unit = {
    this.velocity = new PhysicsVector(
      player.orientation.x * player.strength,
      player.orientation.y * player.strength,
      player.strength
    )
  }
}
```

Objects Review

HealthPotion		
State	location: PhysicsVector	(5.0, 7.0, 0.0)
	velocity: PhysicsVector	(0.0, 0.0, 0.0)
	volume: Int	3
Behavior	use(player: Player): Unit	

```
object HealthPotion {
  var location: PhysicsVector = new PhysicsVector(5.0, 7.0, 0.0)
  var velocity: PhysicsVector = new PhysicsVector(0.0, 0.0, 0.0)
  val volume: Int = 3

  def use(player: Player): Unit = {
    player.health = (player.health + this.volume).min(player.maxHealth)
  }
}
```

Objects Review

- But this is restrictive
- Game can only have one Ball, one HealthPotion, and on Player
- Can play, but not very fun

Player		
State	location: PhysicsVector	(2.0, -2.0, 2.0)
	velocity: PhysicsVector	(0.0, -1.0, 0.0)
	orientation: PhysicsVector	(0.5, -0.5, 0.0)
	health: Int	17
	maxHealth: Int	20
	strength: Int	25
Behavior	useBall(ball: Ball): Unit	
	useHealthPotion(potion: HealthPotion): Unit	

Ball		
State	location: PhysicsVector	(1.0, 5.0, 2.0)
	velocity: PhysicsVector	(1.0, 1.0, 10.0)
	mass: Double	5.0
Behavior	use(player: Player): Unit	

HealthPotion		
State	location: PhysicsVector	(5.0, 7.0, 0.0)
	velocity: PhysicsVector	(0.0, 0.0, 0.0)
	volume: Int	3
Behavior	use(player: Player): Unit	

Classes Review

- This is why we use classes
- Classes let us create multiple objects of type Ball, HealthPotion, and Player

Player	
State	location: PhysicsVector
	velocity: PhysicsVector
	velocity: PhysicsVector
	health: Int
	maxHealth: Int
	strength: Int
Behavior	useBall(ball: Ball): Unit
	useHealthPotion(potion: HealthPotion): Unit

Ball	
State	location: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

HealthPotion	
State	location: PhysicsVector
	velocity: PhysicsVector
	volume: Int
Behavior	use(player: Player): Unit

Classes Review

Player	
State	location: PhysicsVector
	velocity: PhysicsVector
	velocity: PhysicsVector
	health: Int
	maxHealth: Int
	strength: Int
Behavior	useBall(ball: Ball): Unit
	useHealthPotion(potion: HealthPotion): Unit

```
class Player(var location: PhysicsVector,
             var velocity: 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)
  }
}
```

Classes Review

Ball	
State	location: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

```scala
class Ball(var location: PhysicsVector,
           var velocity: PhysicsVector,
           val mass: Double) {

  def use(player: Player): Unit = {
    this.velocity = new PhysicsVector(
      player.orientation.x * player.strength,
      player.orientation.y * player.strength,
      player.strength
    )
  }

}
```

Classes Review

HealthPotion	
State	location: PhysicsVector
	velocity: PhysicsVector
	volume: Int
Behavior	use(player: Player): Unit

```
class HealthPotion(var location: PhysicsVector,
                   var velocity: PhysicsVector,
                   val volume: Int) {

  def use(player: Player): Unit = {
    player.health = (player.health + this.volume).min(player.maxHealth)
  }

}
```

Classes Review

- Use the class to create multiple objects with different states

Ball	
State	location: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

```
var ball1: Ball = new Ball(
  new PhysicsVector(1.0, 5.0, 2.0),
  new PhysicsVector(1.0, 1.0, 10.0),
  5.0
)
// ball1 stores 54224
```

Ball@54224		
State	location: PhysicsVector	(1.0, 5.0, 2.0)
	velocity: PhysicsVector	(1.0, 1.0, 10.0)
	mass: Double	5.0
Behavior	use(player: Player): Unit	

```
var ball2: Ball = new Ball(
  new PhysicsVector(6.0, -3.0, 2.0),
  new PhysicsVector(0.0, 4.5, 4.5),
  10.0
)
// ball2 stores 21374
```

Ball@21374		
State	location: PhysicsVector	(6.0, -3.0, 2.0)
	velocity: PhysicsVector	(0.0, 4.5, 4.5)
	mass: Double	10.0
Behavior	use(player: Player): Unit	

Inheritance

- Use inheritance to create classes with different behavior
- Observe: Ball and HealthPotion have a lot in common

Ball	
State	location: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

HealthPotion	
State	location: PhysicsVector
	velocity: PhysicsVector
	volume: Int
Behavior	use(player: Player): Unit

Inheritance

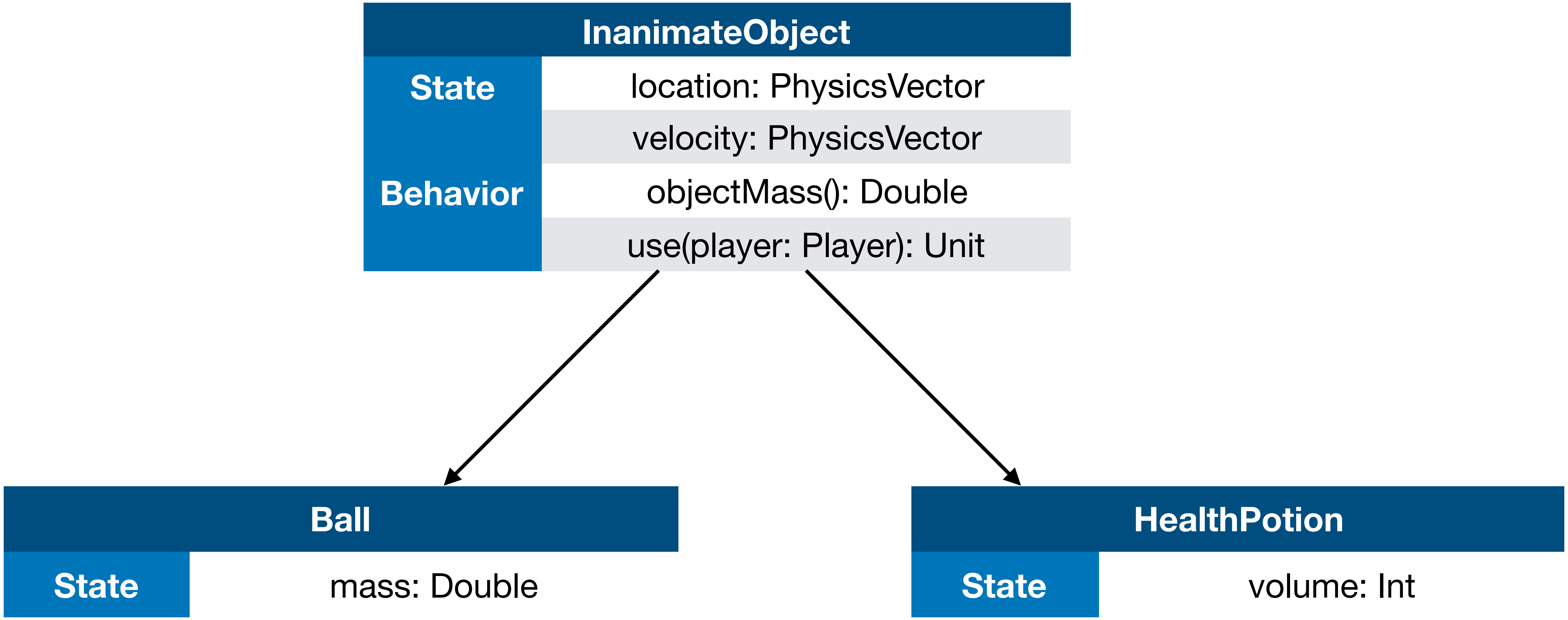
- Can add much more common functionality (that doesn't fit on a slide)
  - Compute mass of a potion based on volume
  - Compute momentum of both types based on mass * velocity
  - Method defining behavior when either hits the ground (bounce or shatter)

Ball	
State	location: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

HealthPotion	
State	location: PhysicsVector
	velocity: PhysicsVector
	volume: Int
Behavior	use(player: Player): Unit

Inheritance

- Factor out common state and behavior into a new class
- Ball and HealthPotion classes **inherent** the state and behavior of InanimateObject
- Ball and HealthPotion add their specific state and behavior



InanimateObject	
State	location: PhysicsVector
	velocity: PhysicsVector
Behavior	objectMass(): Double
	use(player: Player): Unit

Ball	
State	mass: Double

HealthPotion	
State	volume: Int

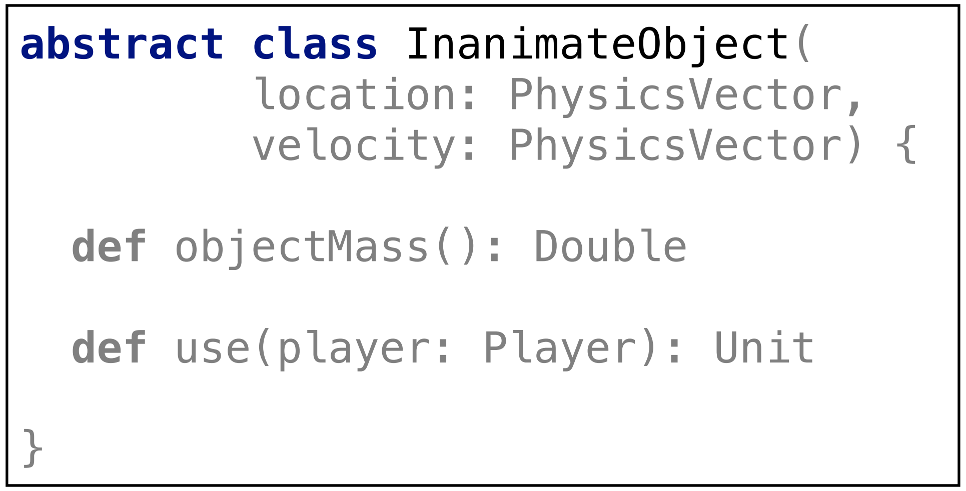
Inheritance

- New class defines what every inheriting class must define
- Any behavior that is to be defined by inheriting classes is declared **abstract**
    - We call this an abstract class
    - Cannot create objects of abstract types
- Inheriting classes will define all abstract behavior
    - We call these concrete classes

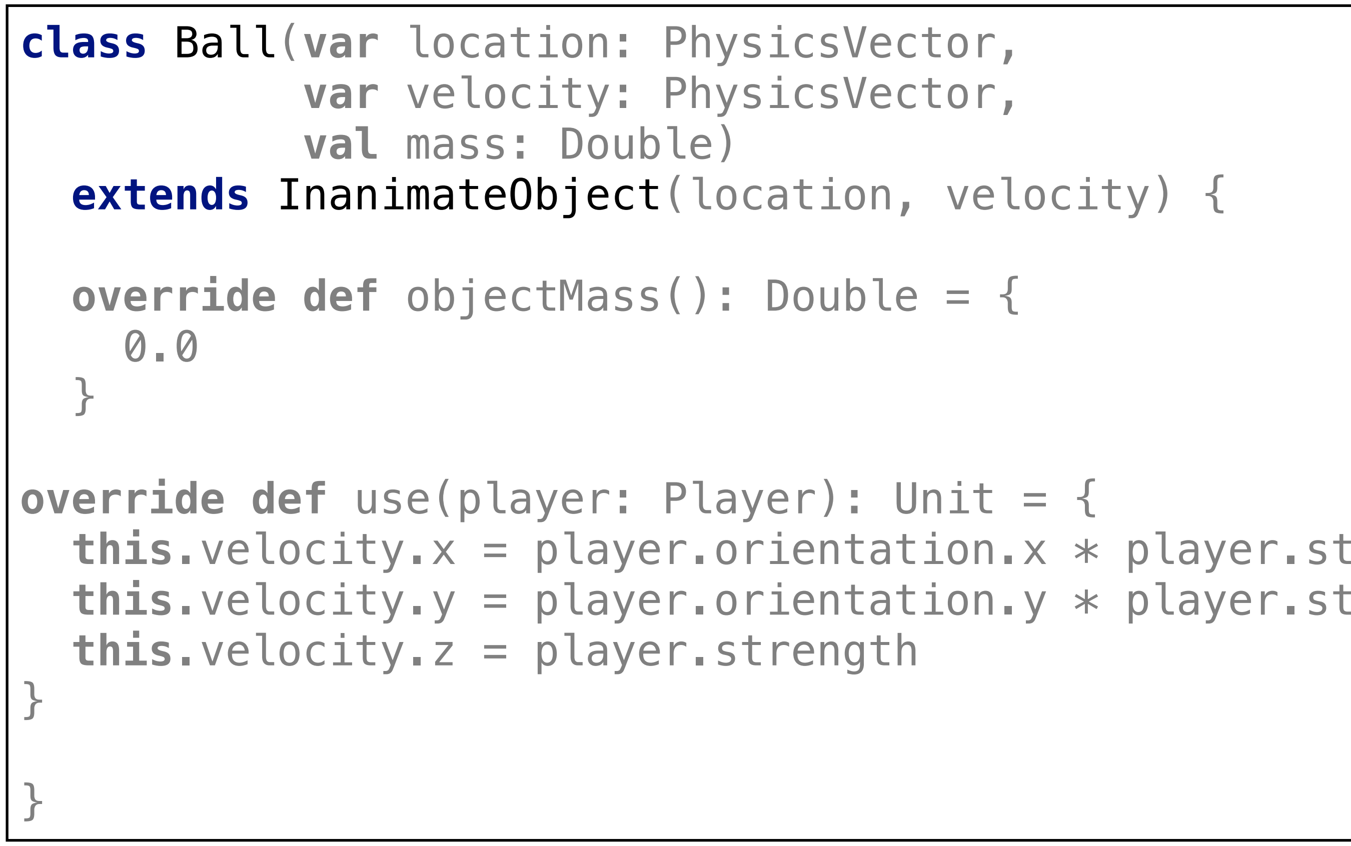
```
abstract class InanimateObject(location: PhysicsVector, velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

}
```

Inheritance

- Use the extends keyword to inherent another class
  - Extend the definition of InanimateObject
  - We call InanimateObject the superclass of Ball



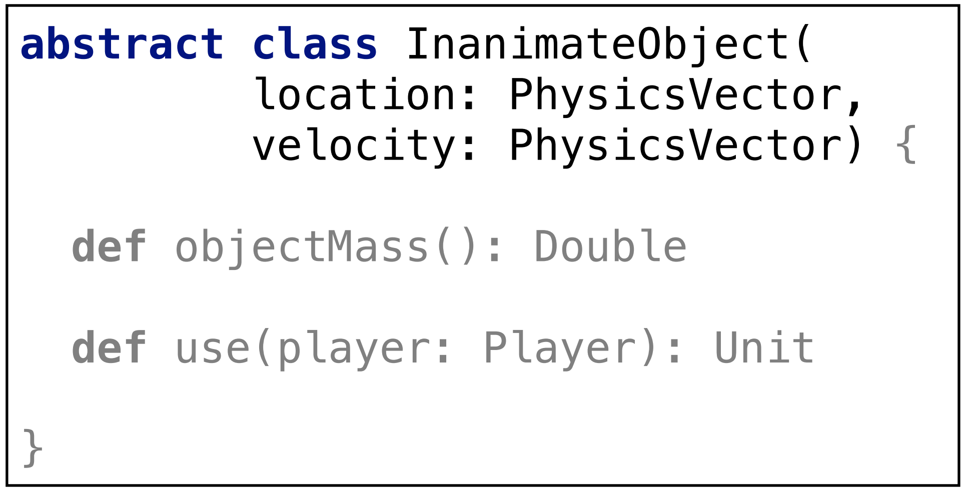
```
abstract class InanimateObject(
         location: PhysicsVector,
         velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

}
```



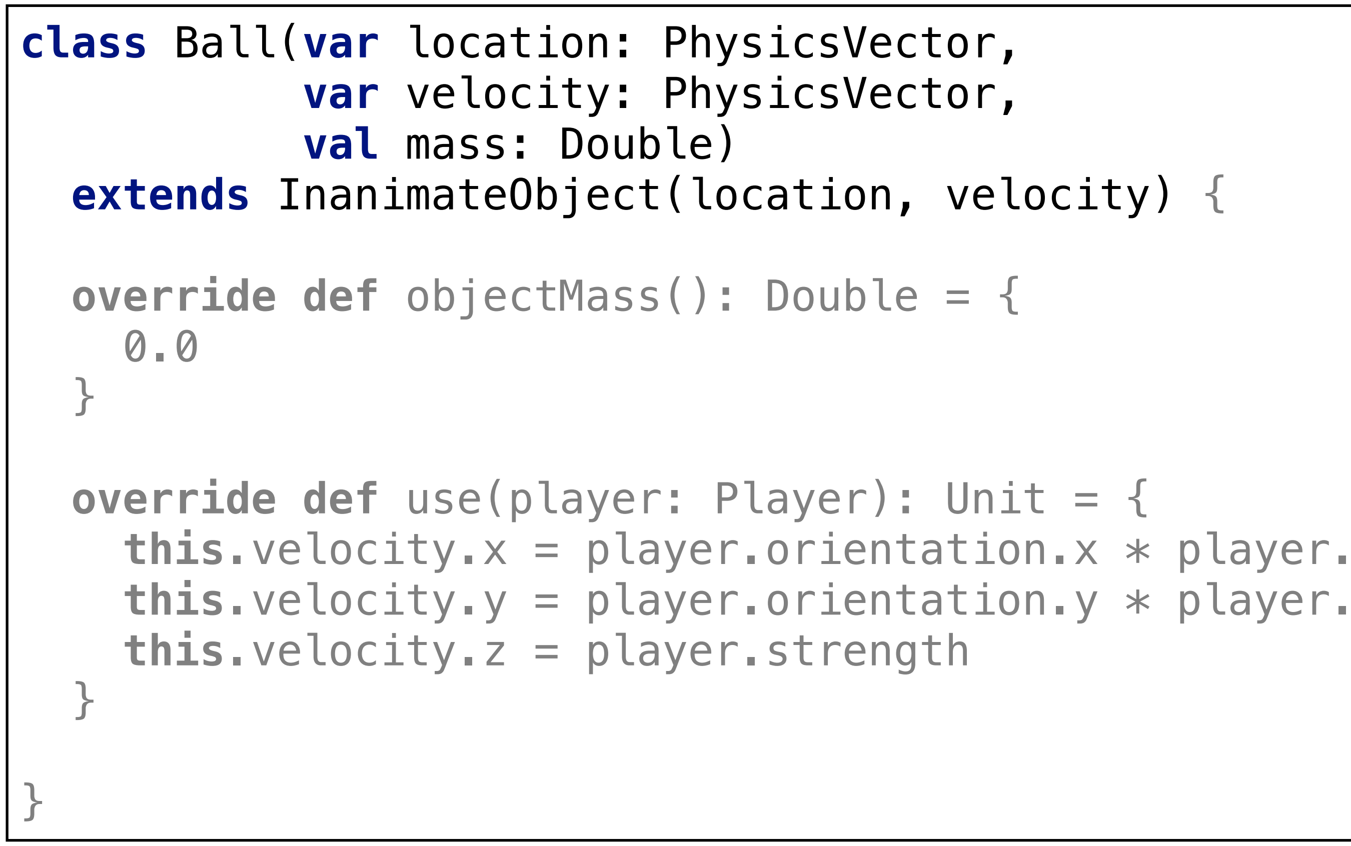
```
class Ball(var location: PhysicsVector,
           var velocity: PhysicsVector,
           val mass: Double)
  extends InanimateObject(location, velocity) {

  override def objectMass(): Double = {
    0.0
  }

override def use(player: Player): Unit = {
  this.velocity.x = player.orientation.x * player.st
  this.velocity.y = player.orientation.y * player.st
  this.velocity.z = player.strength
}

}
```

Inheritance

- Ball has it's own constructor
- Ball must call InanimateObject's constructor
- var/val declared in concrete class to make these public



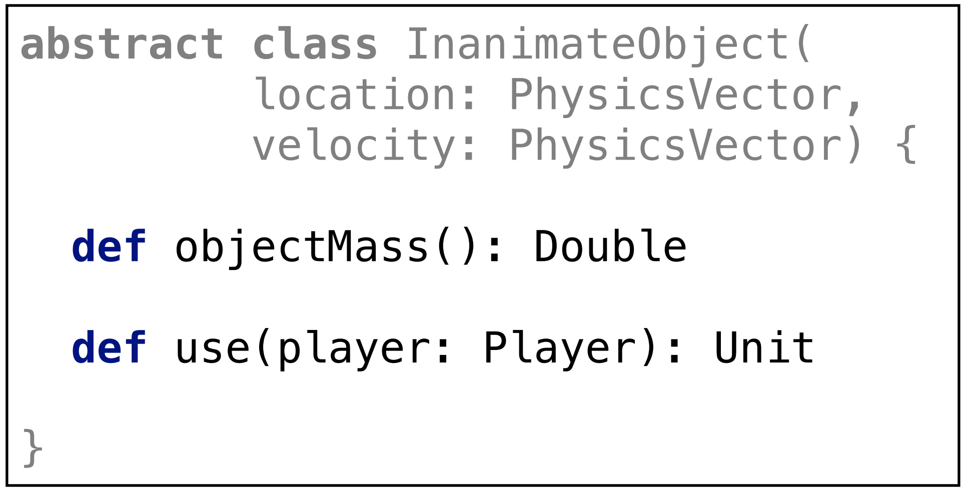
```
abstract class InanimateObject(
         location: PhysicsVector,
         velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

}
```



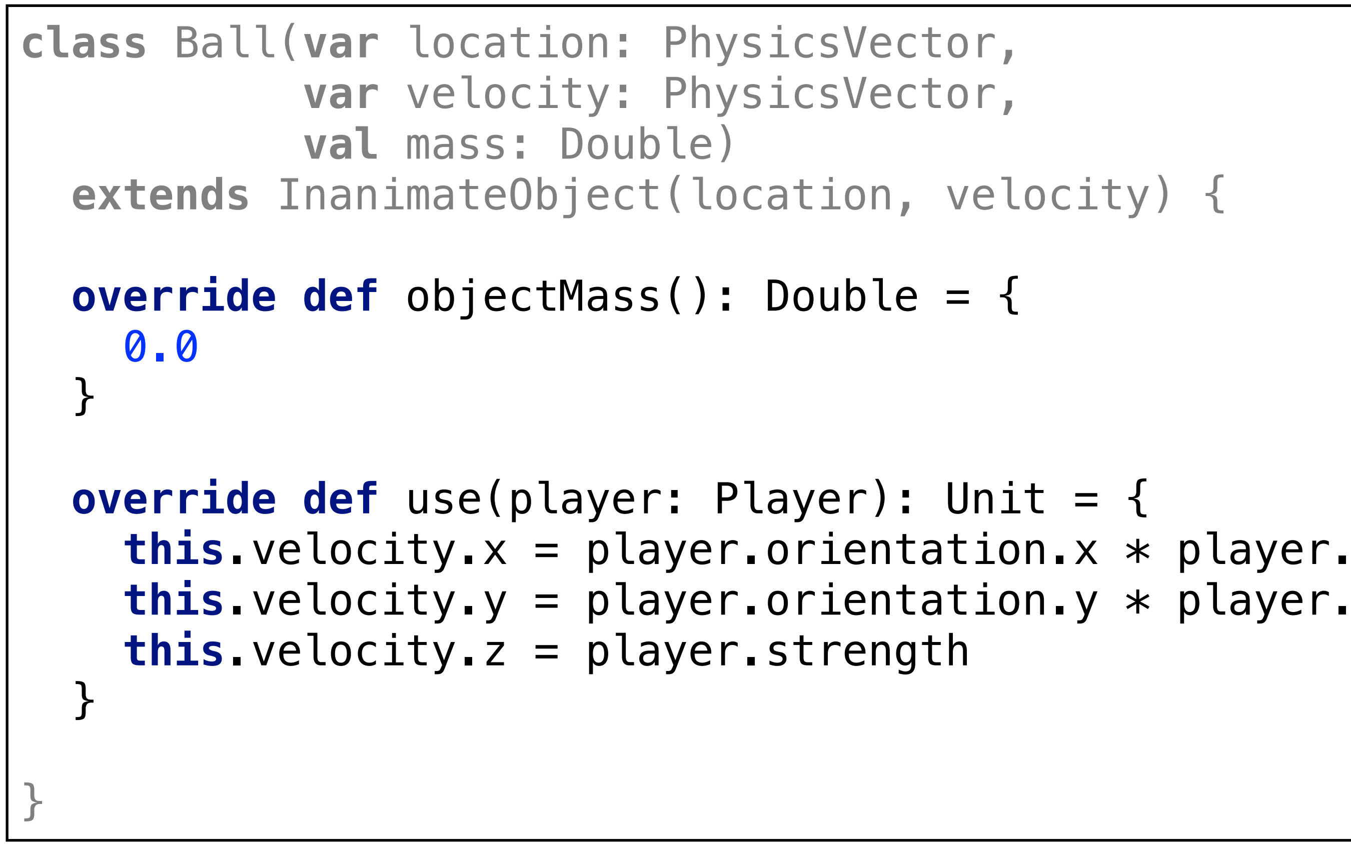
```
class Ball(var location: PhysicsVector,
           var velocity: PhysicsVector,
           val mass: Double)
  extends InanimateObject(location, velocity) {

  override def objectMass(): Double = {
    0.0
  }

  override def use(player: Player): Unit = {
    this.velocity.x = player.orientation.x * player.
    this.velocity.y = player.orientation.y * player.
    this.velocity.z = player.strength
  }

}
```

Inheritance

- Implement all abstract behavior
- Use the **override** keyword when overwriting behavior from the superclass
- Override all abstract methods with behavior for this class



```scala
abstract class InanimateObject(
         location: PhysicsVector,
         velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

}
```



```scala
class Ball(var location: PhysicsVector,
           var velocity: PhysicsVector,
           val mass: Double)
  extends InanimateObject(location, velocity) {

  override def objectMass(): Double = {
    0.0
  }

  override def use(player: Player): Unit = {
    this.velocity.x = player.orientation.x * player.
    this.velocity.y = player.orientation.y * player.
    this.velocity.z = player.strength
  }

}
```

Inheritance

- Define different behavior for each base class
- Define similar types with some difference

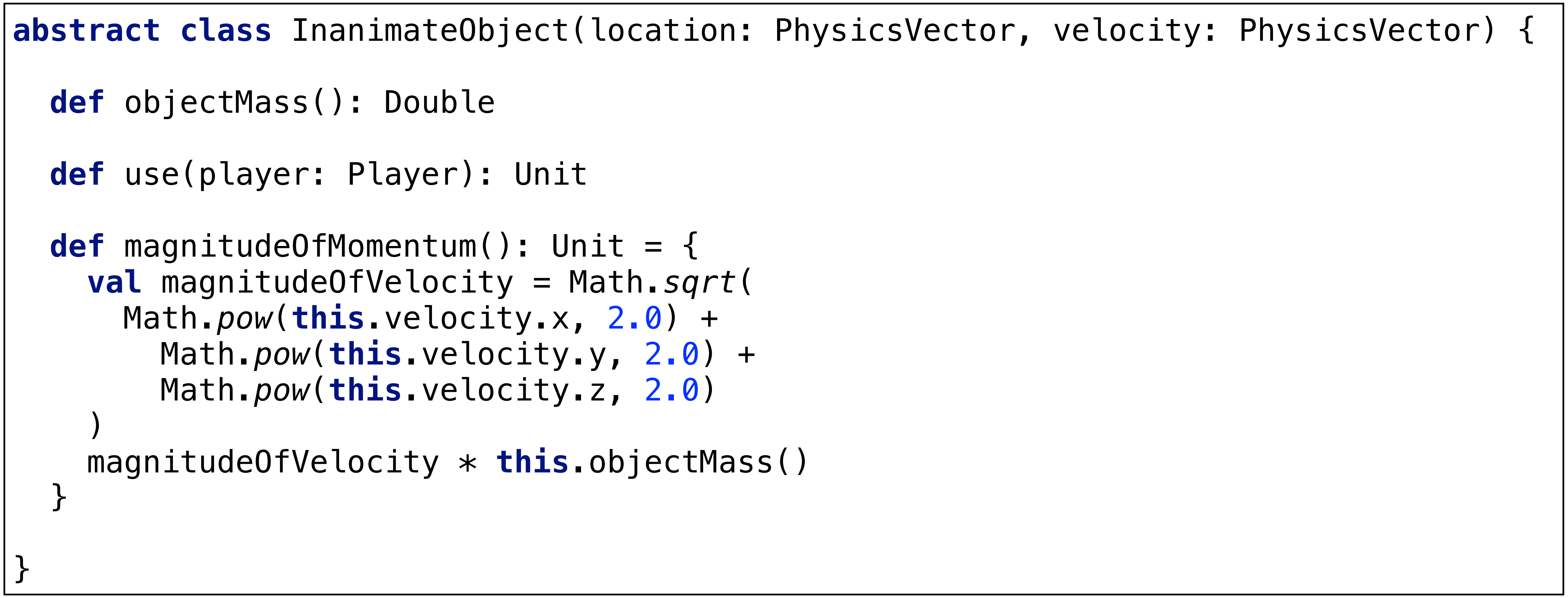
```
abstract class InanimateObject(
         location: PhysicsVector,
         velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit
}
```

```
class HealthPotion(var location: PhysicsVector,
                   var velocity: PhysicsVector,
                   val volume: Int)
  extends InanimateObject(location, velocity) {


  override def objectMass(): Double = {
    val massPerVolume: Double = 7.0
    volume * massPerVolume
  }
  def use(player: Player): Unit = {
    player.health = (player.health +
                  this.volume).min(player.maxHealth)
  }
}
```

```
class Ball(var location: PhysicsVector,
           var velocity: PhysicsVector,
           val mass: Double)
  extends InanimateObject(location, velocity) {

  override def objectMass(): Double = {
    0.0
  }

  override def use(player: Player): Unit = {
    this.velocity.x = player.orientation.x * player.
    this.velocity.y = player.orientation.y * player.
    this.velocity.z = player.strength
  }

}
```

Inheritance

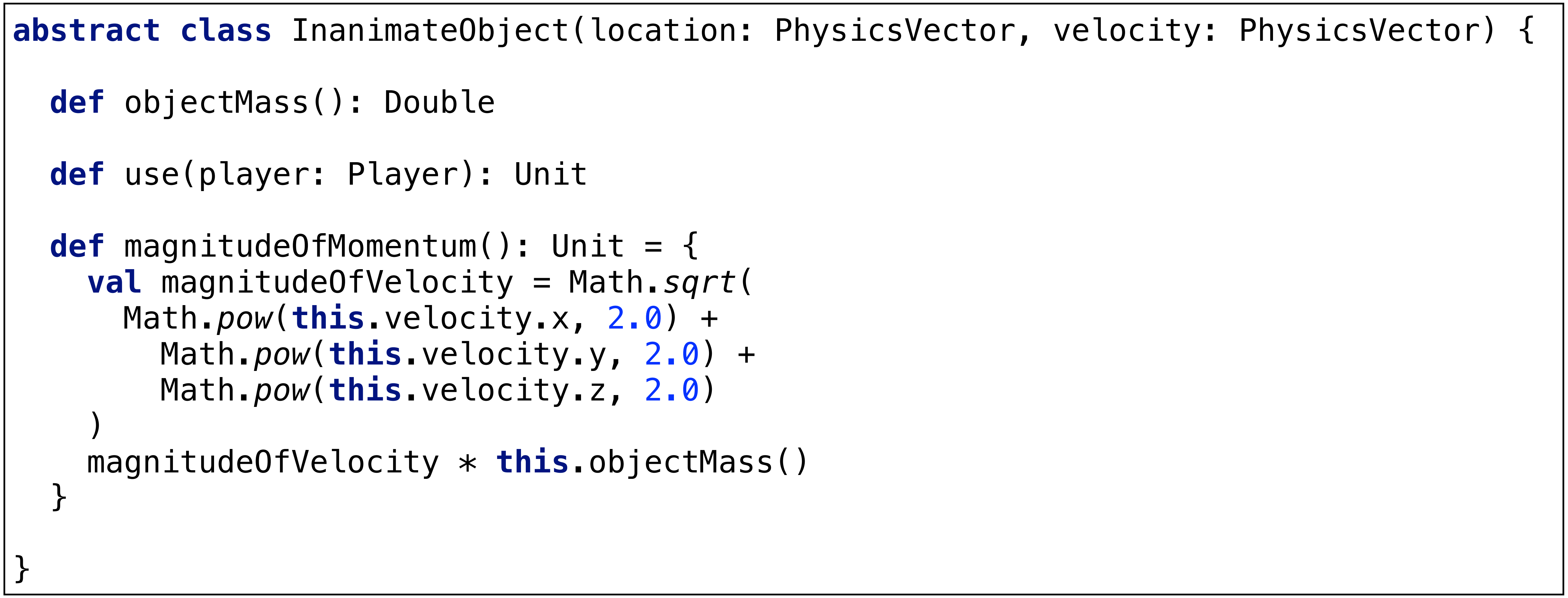
- **OK, BUT Y THO?**
- Add behavior to InanimateObject
- Behavior is added to ALL inheriting classes



```
abstract class InanimateObject(location: PhysicsVector, velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Unit = {
    val magnitudeOfVelocity = Math.sqrt(
      Math.pow(this.velocity.x, 2.0) +
        Math.pow(this.velocity.y, 2.0) +
        Math.pow(this.velocity.z, 2.0)
    )
    magnitudeOfVelocity * this.objectMass()
  }

}
```

Inheritance

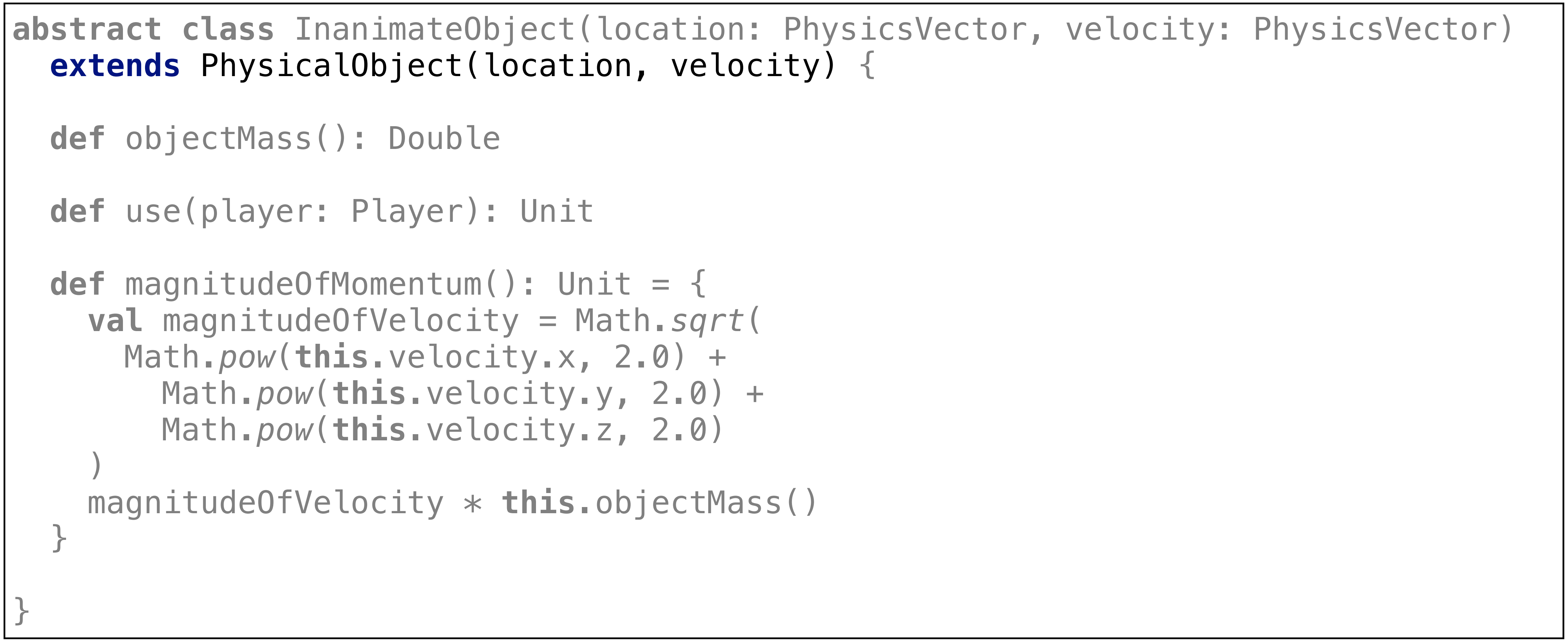
- We may want many, many more subtypes of InanimateObjects in our game
- Any common functionality added to InanimateObject
  - Easy to add functionality to ALL subtypes will very little effort



```
abstract class InanimateObject(location: PhysicsVector, velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Unit = {
    val magnitudeOfVelocity = Math.sqrt(
      Math.pow(this.velocity.x, 2.0) +
        Math.pow(this.velocity.y, 2.0) +
        Math.pow(this.velocity.z, 2.0)
    )
    magnitudeOfVelocity * this.objectMass()
  }

}
```

Inheritance

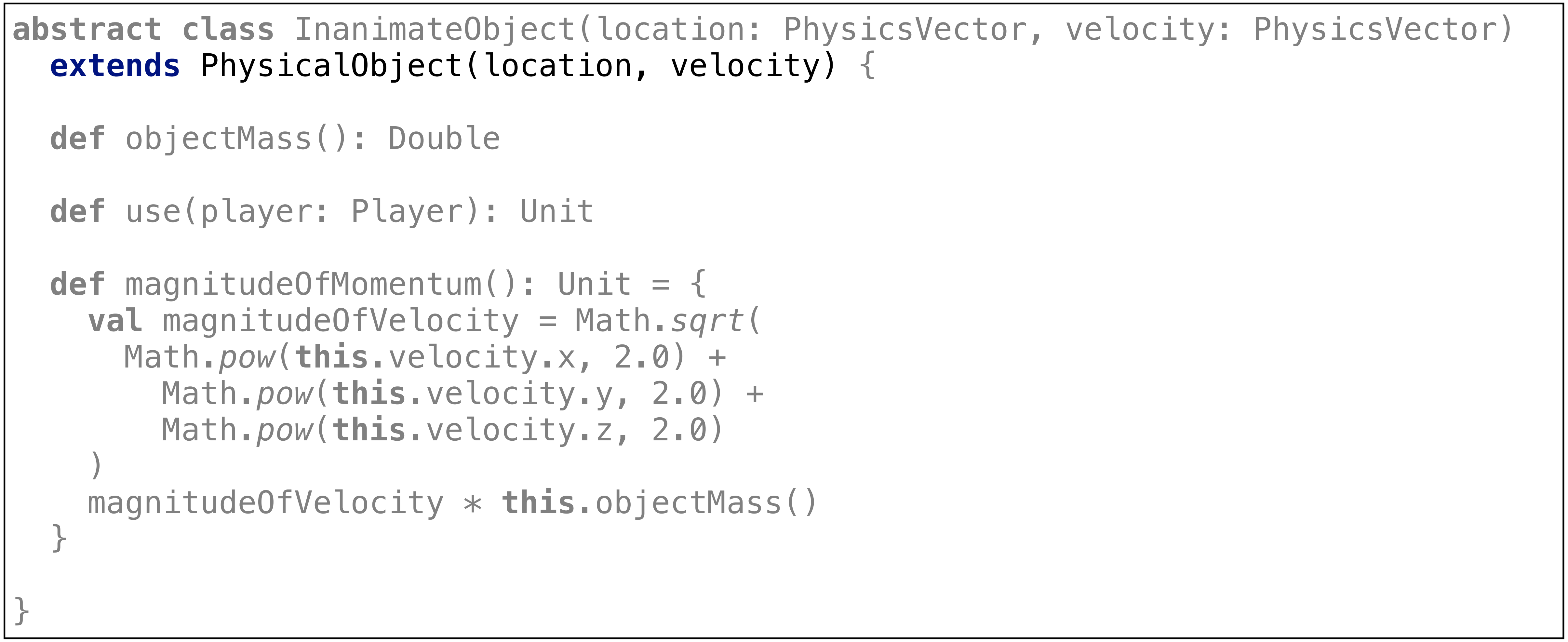
- **But wait!**
  - **There's more**



```scala
abstract class InanimateObject(location: PhysicsVector, velocity: PhysicsVector)
  extends PhysicalObject(location, velocity) {

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Unit = {
    val magnitudeOfVelocity = Math.sqrt(
      Math.pow(this.velocity.x, 2.0) +
        Math.pow(this.velocity.y, 2.0) +
        Math.pow(this.velocity.z, 2.0)
    )
    magnitudeOfVelocity * this.objectMass()
  }

}
```

Inheritance

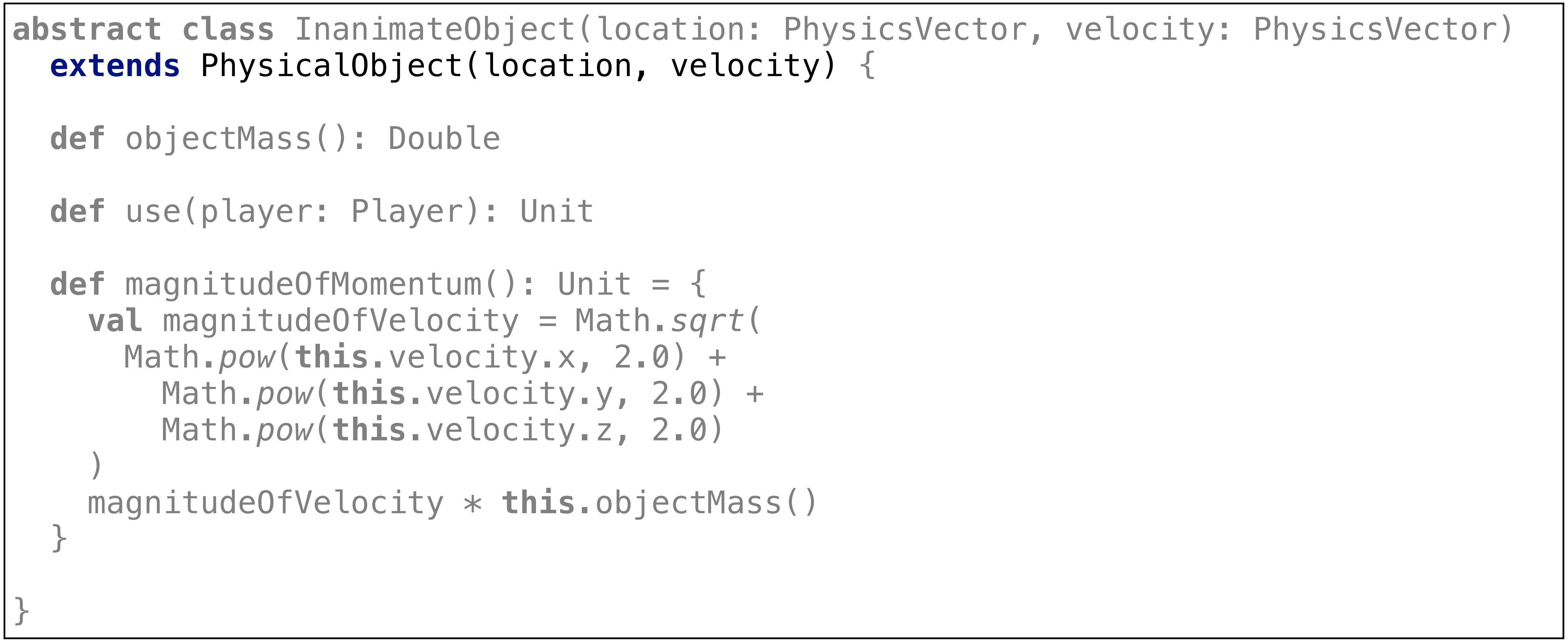
- If we want Ball, HealthPotion, and all other InanimateObjects to work with our physics engine
- Extend PhysicalObject



```
abstract class InanimateObject(location: PhysicsVector, velocity: PhysicsVector)
  extends PhysicalObject(location, velocity) {

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Unit = {
    val magnitudeOfVelocity = Math.sqrt(
      Math.pow(this.velocity.x, 2.0) +
        Math.pow(this.velocity.y, 2.0) +
        Math.pow(this.velocity.z, 2.0)
    )
    magnitudeOfVelocity * this.objectMass()
  }

}
```

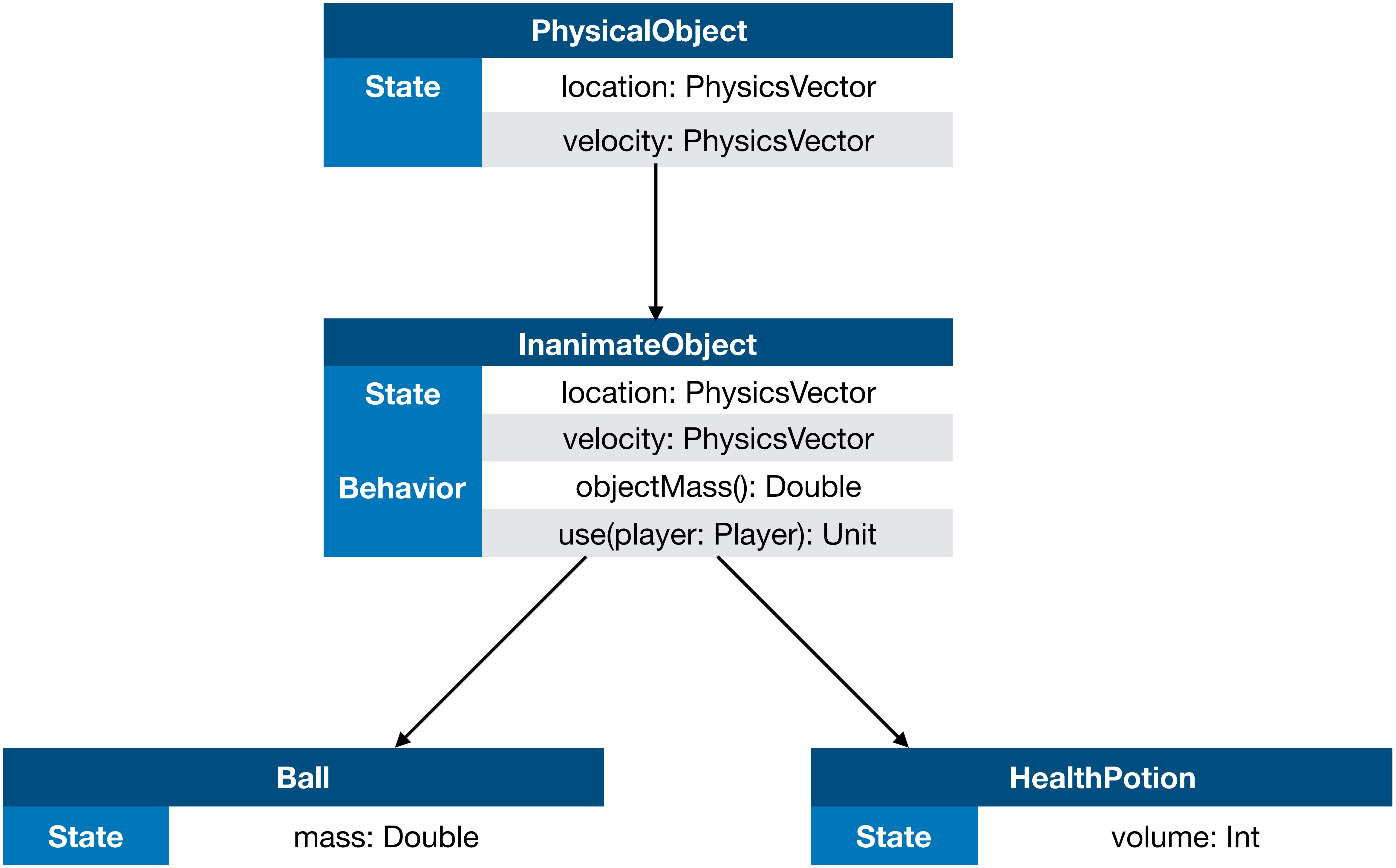
Inheritance

- If we want Ball, HealthPotion, and all other InanimateObjects to work with our physics engine
- Extend PhysicalObject



```
abstract class InanimateObject(location: PhysicsVector, velocity: PhysicsVector)
  extends PhysicalObject(location, velocity) {

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Unit = {
    val magnitudeOfVelocity = Math.sqrt(
      Math.pow(this.velocity.x, 2.0) +
        Math.pow(this.velocity.y, 2.0) +
        Math.pow(this.velocity.z, 2.0)
    )
    magnitudeOfVelocity * this.objectMass()
  }

}
```

Inheritance



PhysicalObject

State

location: PhysicsVector

velocity: PhysicsVector

InanimateObject

State

location: PhysicsVector

velocity: PhysicsVector

Behavior

objectMass(): Double

use(player: Player): Unit

Ball

State

mass: Double

HealthPotion

State

volume: Int

Lecture Question

Objective: Study the syntax of inheritance in Scala

Question: [Scala] In a package named "inheritance" create an abstract class named "Animal" and concrete classes named "Cat" and "Dog". Implement the following in each class:

Animal:

- A constructor that takes a String called name (Do not use either val or var. It will be declared in the base classes)
- An abstract method named sound that takes no parameters and returns a String

Cat:

- Inherent Animal
- A constructor that take a String called name as a value (use val to declare name)
- Override sound() to return "meow"

Dog:

- Inherent Animal
- A constructor that take a String called name as a value (use val to declare name)
- Override sound() to return "woof"

* This question will be open until midnight