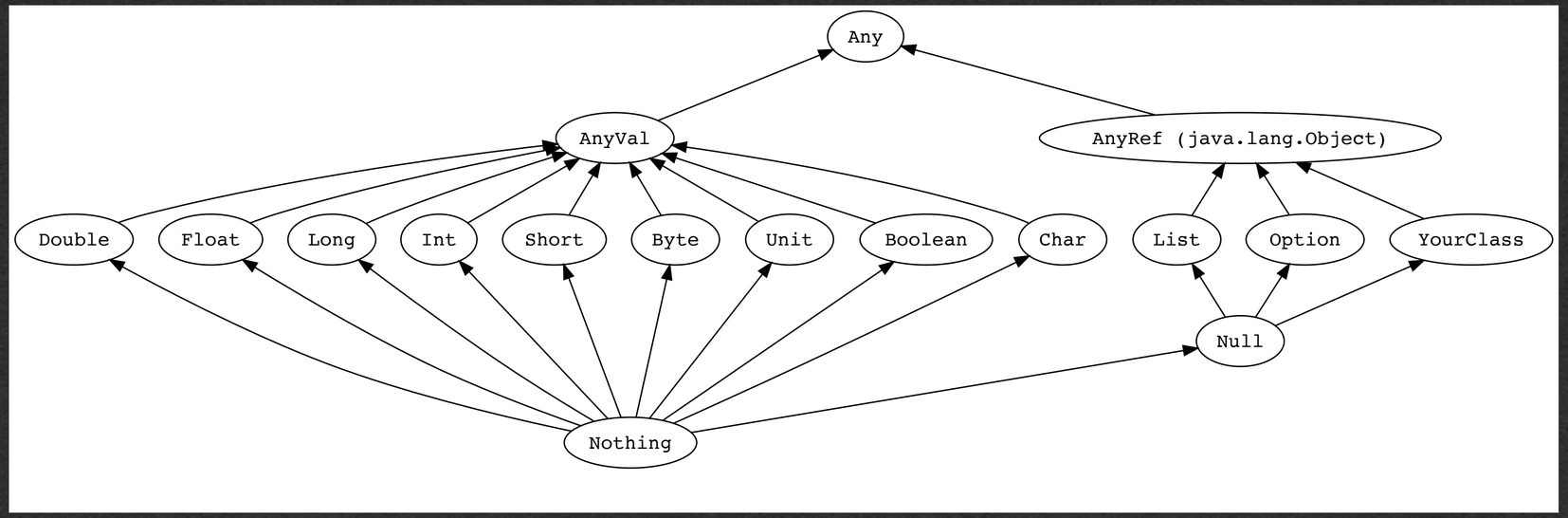
Inheritance

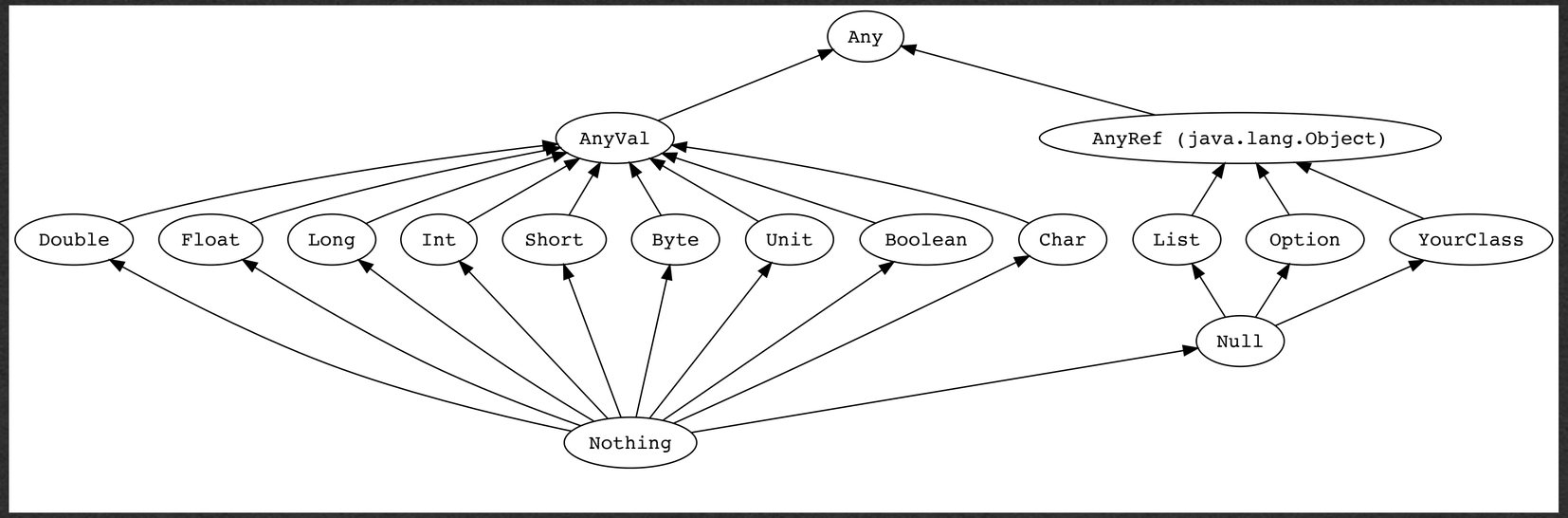
Scala Type Hierarchy



- All objects share Any as their base types
- Classes extending **AnyVal** will be stored on the **stack**
- Classes extending **AnyRef** will be stored on the **heap**

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

Scala Type Hierarchy



- But what does it mean for a class to **extend** another class?
  - This is **inheritance**

- Let's explore this concept through an example

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

Overview

- Suppose we're making a game and we want various objects that will interact with each other
- We'll setup a simple game where:
  - Players can move in a 2d top-down space
  - Each player has a set health and strength
  - Players can pick up and throw DodgeBalls
  - If a player gets hit with a DodgeBall, they lose health
  - Players can collect health potions to regain health
- Note: We won't build this full game, but we will build some of the game mechanics today

Objects Review

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

Objects Review

- We'll use a PhysicsVector class throughout this example
- Represents a vector in a 3d space
- Used to store the location of objects in the game
- Each parameter has a default value of 0.0
  - Ex. **new** PhysicsVector(2.0, -2.0) creates the vector (2.0, -2.0, 0.0)

```
class PhysicsVector(var x: Double = 0.0, var y: Double = 0.0, var z: Double = 0.0) {}
```

Objects Review

Player		
State	playerLocation: PhysicsVector	(2.0, -2.0)
	orientation: PhysicsVector	(0.5, -0.5)
	health: Int	17
	maxHealth: Int	20
	strength: Int	25
Behavior	useDodgeBall(DodgeBall: DodgeBall): Unit	
	useHealthPotion(potion: HealthPotion): Unit	

```
object Player {
  var playerLocation: PhysicsVector = new PhysicsVector(2.0, -2.0)
  var orientation: PhysicsVector = new PhysicsVector(0.5, -0.5)

  var health: Int = 17
  val maxHealth: Int = 20

  val strength: Int = 25


  def useDodgeBall(dodgeBall: DodgeBall): Unit = {
    dodgeBall.use(this)
  }

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

Objects Review

DodgeBall		
State	dbLocation: PhysicsVector	(1.0, 5.0)
	velocity: PhysicsVector	(0.0, 0.0)
	mass: Double	5.0
Behavior	use(player: Player): Unit	

```
object DodgeBall {
  var dbLocation: PhysicsVector = new PhysicsVector(1.0, 5.0)
  var velocity: PhysicsVector = new PhysicsVector(0.0, 0.0)
  val mass: Double = 5.0

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

Objects Review

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

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

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

Objects Review

- This is very restrictive!
- Game can only have one DodgeBall, one HealthPotion, and one Player
- Can play, but not very fun

Player		
State	playerLocation: PhysicsVector	(2.0, -2.0)
	orientation: PhysicsVector	(0.5, -0.5)
	health: Int	17
	maxHealth: Int	20
	strength: Int	25
Behavior	useDodgeBall(DodgeBall: DodgeBall): Unit	
	useHealthPotion(potion: HealthPotion): Unit	

DodgeBall		
State	dbLocation: PhysicsVector	(1.0, 5.0)
	velocity: PhysicsVector	(0.0, 0.0)
	mass: Double	5.0
Behavior	use(player: Player): Unit	

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

Classes Review

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

Player	
State	playerLocation: PhysicsVector
	orientation: PhysicsVector
	health: Int
	maxHealth: Int
	strength: Int
Behavior	useDodgeBall(DodgeBall: DodgeBall): Unit
	useHealthPotion(potion: HealthPotion): Unit

DodgeBall	
State	dbLocation: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

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

Classes Review

Player	
State	playerLocation: PhysicsVector
	orientation: PhysicsVector
	health: Int
	maxHealth: Int
	strength: Int
Behavior	useDodgeBall(DodgeBall: DodgeBall): Unit
	useHealthPotion(potion: HealthPotion): Unit

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

  var health: Int = maxHealth

  def useDodgeBall(DodgeBall: DodgeBall): Unit = {
    DodgeBall.use(this)
  }

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

Classes Review

DodgeBall	
State	dbLocation: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

```
class DodgeBall(var dbLocation: 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
    )
  }

}
```

Classes Review

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

```
class HealthPotion(var potionLocation: 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

DodgeBall	
State	dbLocation: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

```
var dodgeBall1: DodgeBall = new DodgeBall(
  new PhysicsVector(1.0, 5.0),
  new PhysicsVector(1.0, 1.0),
  5.0
)
// dodgeBall1 stores the reference 0x200
```

DodgeBall 0x200		
	dbLocation: PhysicsVector	(1.0, 5.0)
	velocity: PhysicsVector	(1.0, 1.0)
	mass: Double	5.0
Behavior	use(player: Player): Unit	

```
var dodgeBall2: DodgeBall = new DodgeBall(
  new PhysicsVector(6.0, -3.0),
  new PhysicsVector(0.0, 4.5),
  10.0
)
// dodgeBall2 stores the reference 0x150
```

DodgeBall 0x150		
State	dbLocation: PhysicsVector	(6.0, -3.0)
	velocity: PhysicsVector	(0.0, 4.5)
	mass: Double	10.0
Behavior	use(player: Player): Unit	

Inheritance

Inheritance

- Use inheritance to create classes with similar state and behavior
- Observe: DodgeBall and HealthPotion have a lot in common

DodgeBall	
State	dbLocation: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

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

Inheritance

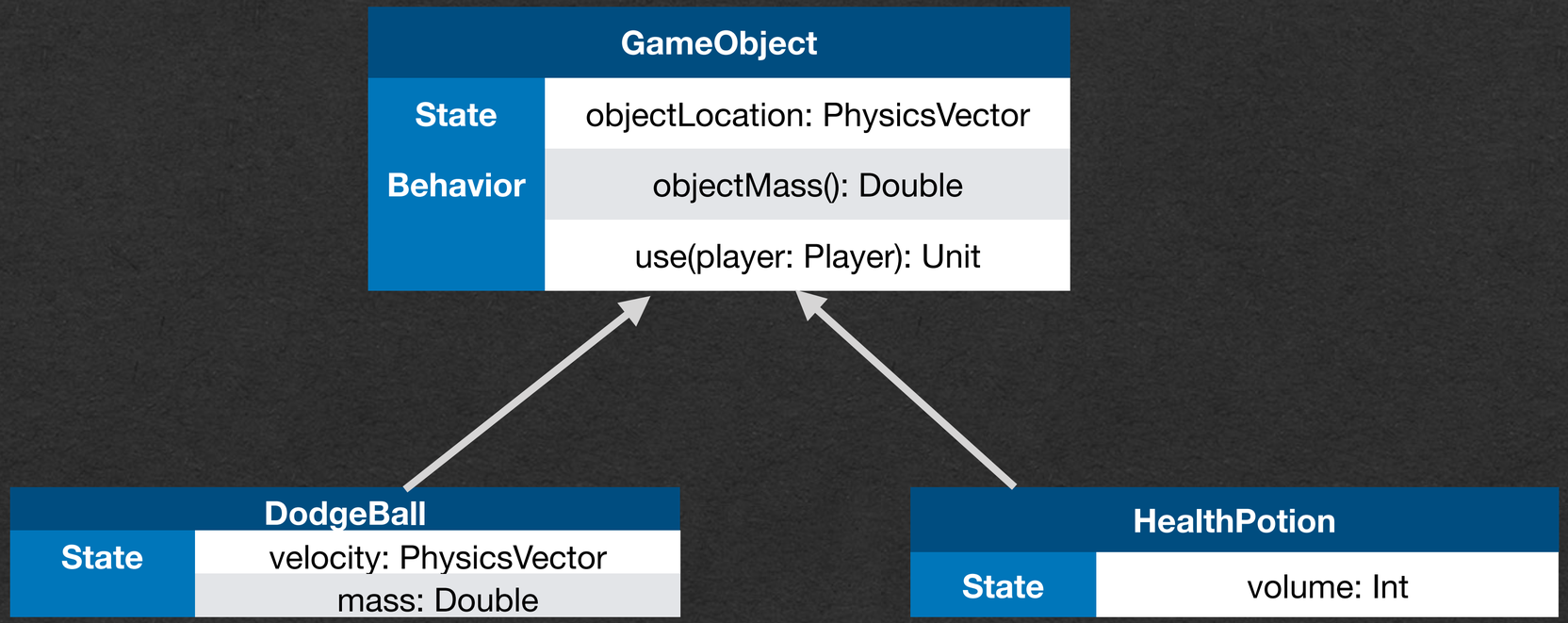
- Can add much more common functionality
  - Compute mass of a potion based on volume
  - Compute momentum of both types based on mass * velocity
  - Method defining behavior when either hits the ground (bounce or shatter)

DodgeBall	
State	dbLocation: PhysicsVector
	velocity: PhysicsVector
	mass: Double
Behavior	use(player: Player): Unit

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

Inheritance

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



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 GameObject(objectLocation: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

}
```

Inheritance

- Abstract methods have no definitions
  - No body
- These methods must be defined by the inheriting classes (dodgeball and health potion)

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

  def objectMass(): Double

  def use(player: Player): Unit

}
```

Inheritance

- Use the extends keyword to inherent another class
  - Extend the definition of GameObject
  - DodgeBall "inherits" all of the state and behavior from GameObject

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

  def objectMass(): Double

  def use(player: Player): Unit

}
```

```
class DodgeBall(dbLocation: PhysicsVector,
                var velocity: PhysicsVector,
                val mass: Double)
  extends GameObject(dbLocation) {

  override def objectMass(): Double = {
    this.mass
  }

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

}
```

Inheritance

- DodgeBall has its own constructor
- DodgeBall must call GameObject's constructor
- var/val declared in concrete class to make these public
- If reusing variable names, only one can be declared with val/var (ie. Cannot have both locations as vars)
  - **Just don't reuse variable names!!**

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

  def objectMass(): Double

  def use(player: Player): Unit

}
```

```
class DodgeBall(dbLocation: PhysicsVector,
                var velocity: PhysicsVector,
                val mass: Double)
  extends GameObject(dbLocation) {

  override def objectMass(): Double = {
    this.mass
  }

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

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

  def objectMass(): Double

  def use(player: Player): Unit

}
```

```
class DodgeBall(dbLocation: PhysicsVector,
                var velocity: PhysicsVector,
                val mass: Double)
  extends GameObject(dbLocation) {

  override def objectMass(): Double = {
    this.mass
  }

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

}
```

Inheritance

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

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

  override def objectMass(): Double = {
    val massPerVolume: Double = 7.0
    volume * massPerVolume
  }

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

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

  def objectMass(): Double

  def use(player: Player): Unit

}
```

```
class DodgeBall(dbLocation: PhysicsVector,
                var velocity: PhysicsVector,
                val mass: Double)
  extends GameObject(dbLocation) {

  override def objectMass(): Double = {
    this.mass
  }

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

}
```

Inheritance

- Example: Calling objectMass() will have different behavior depending on the **type** of the object

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

  def objectMass(): Double

  def use(player: Player): Unit

}
```

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

  override def objectMass(): Double = {
    val massPerVolume: Double = 7.0
    volume * massPerVolume
  }

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

```
class DodgeBall(dbLocation: PhysicsVector,
                var velocity: PhysicsVector,
                val mass: Double)
  extends GameObject(dbLocation) {

  override def objectMass(): Double = {
    this.mass
  }

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

}
```

Inheritance

- OK, BUT Y THO?

Inheritance

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

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

  def objectMass(): Double

  def use(player: Player): Unit

  def closeToPlayer(player: Player): Boolean = {
    val distanceToPlayer = Math.sqrt(
      Math.pow(this.location.x - player.location.x, 2.0) +
        Math.pow(this.location.y - player.location.y, 2.0)
    )
    val threshold: Double = 0.5
    distanceToPlayer < threshold
  }

}
```

Inheritance

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

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

  def objectMass(): Double

  def use(player: Player): Unit

  def closeToPlayer(player: Player): Boolean = {
    val distanceToPlayer = Math.sqrt(
      Math.pow(this.location.x - player.location.x, 2.0) +
        Math.pow(this.location.y - player.location.y, 2.0)
    )
    val threshold: Double = 0.5
    distanceToPlayer < threshold
  }

}
```

Inheritance

- But wait!
- There's more

```
abstract class PhysicsObject(var location: PhysicsVector) {

  // Physics functionality not implemented

}
```

```
abstract class GameObject(objectLocation: PhysicsVector) extends PhysicsObject(objectLocation) {

  def objectMass(): Double

  def use(player: Player): Unit

  def closeToPlayer(player: Player): Boolean = {
    val distanceToPlayer = Math.sqrt(
      Math.pow(this.location.x - player.location.x, 2.0) +
        Math.pow(this.location.y - player.location.y, 2.0)
    )
    val threshold: Double = 0.5
    distanceToPlayer < threshold
  }

}
```

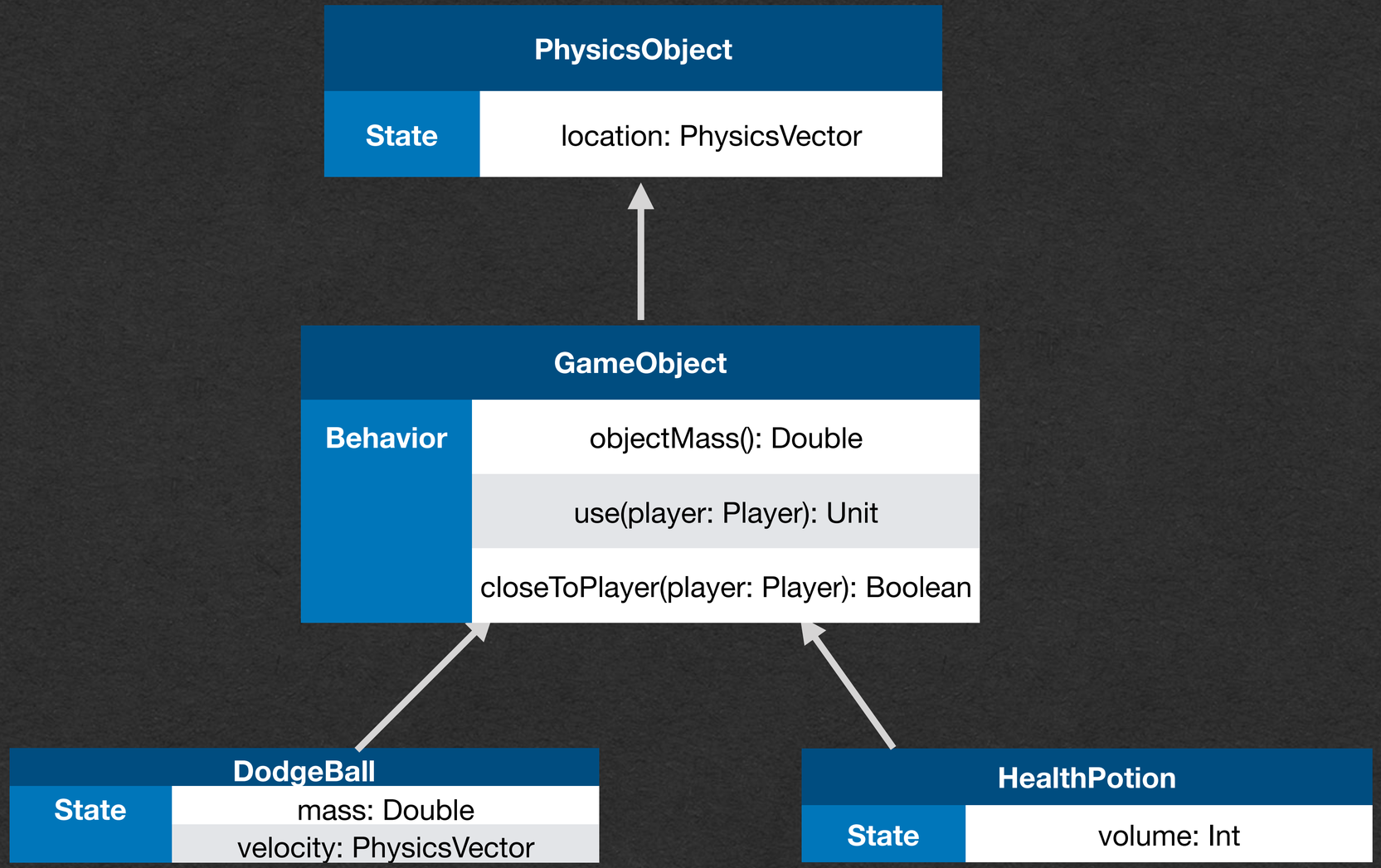
Inheritance

- Suppose we have a physics engine that apply physics to objects of type PhysicsObject
- We want to use this physics engine to control the movement of DodgeBalls and HealthPotions
- Solution: Have GameObject extend PhysicsObject!

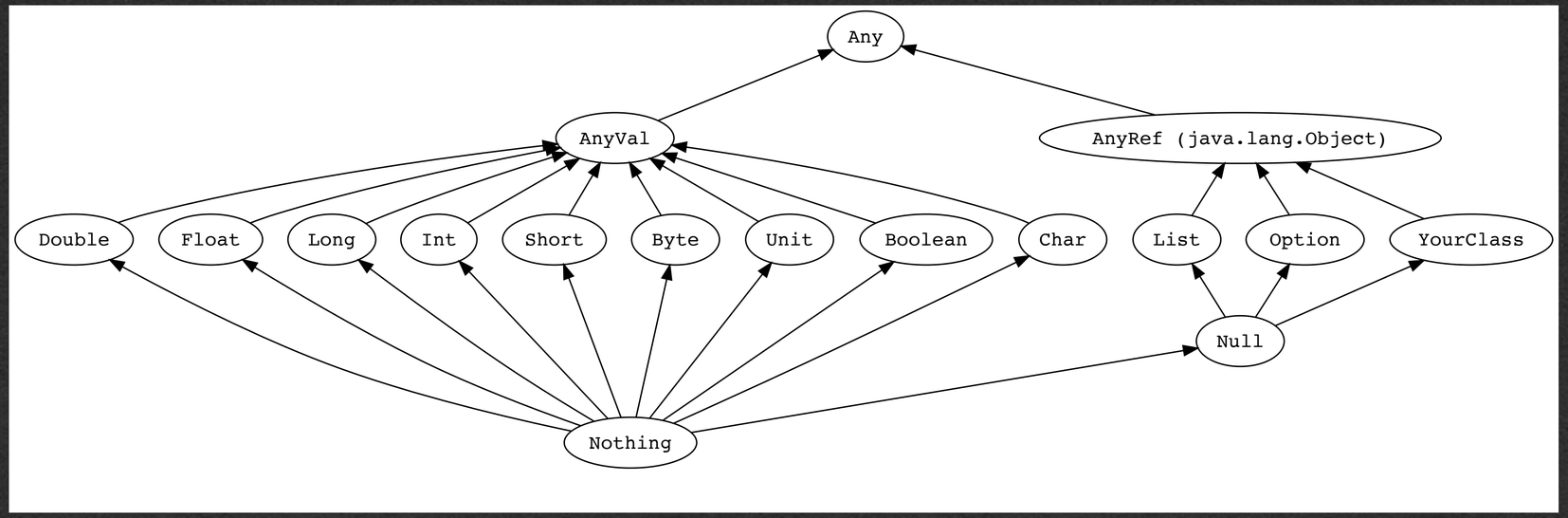
```
abstract class PhysicsObject(var location: PhysicsVector) {

  // Physics functionality not implemented

}
```

```
abstract class GameObject(objectLocation: PhysicsVector) extends PhysicsObject(objectLocation) {

  def objectMass(): Double

  def use(player: Player): Unit

  def closeToPlayer(player: Player): Boolean = {
    val distanceToPlayer = Math.sqrt(
      Math.pow(this.location.x - player.location.x, 2.0) +
        Math.pow(this.location.y - player.location.y, 2.0)
    )
    val threshold: Double = 0.5
    distanceToPlayer < threshold
  }

}
```

Inheritance



Scala Type Hierarchy



- All objects share Any as their base types
- Classes extending **AnyVal** will be stored on the **stack**
- Classes extending **AnyRef** will be stored on the **heap**

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