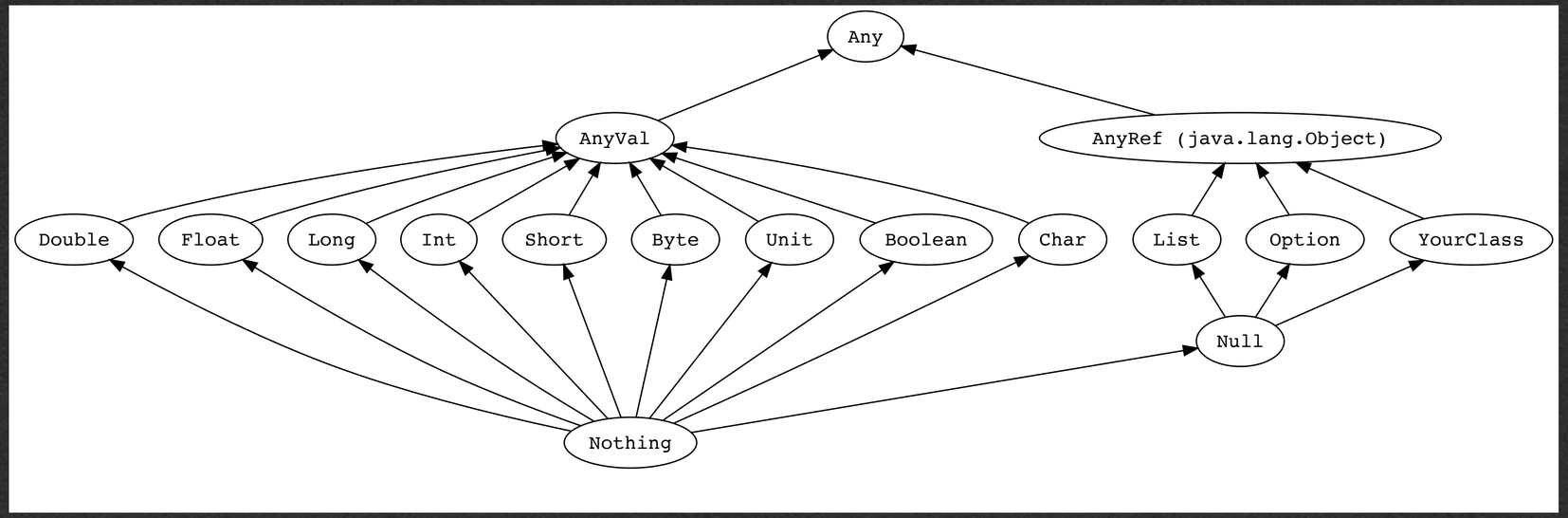
Polymorphism

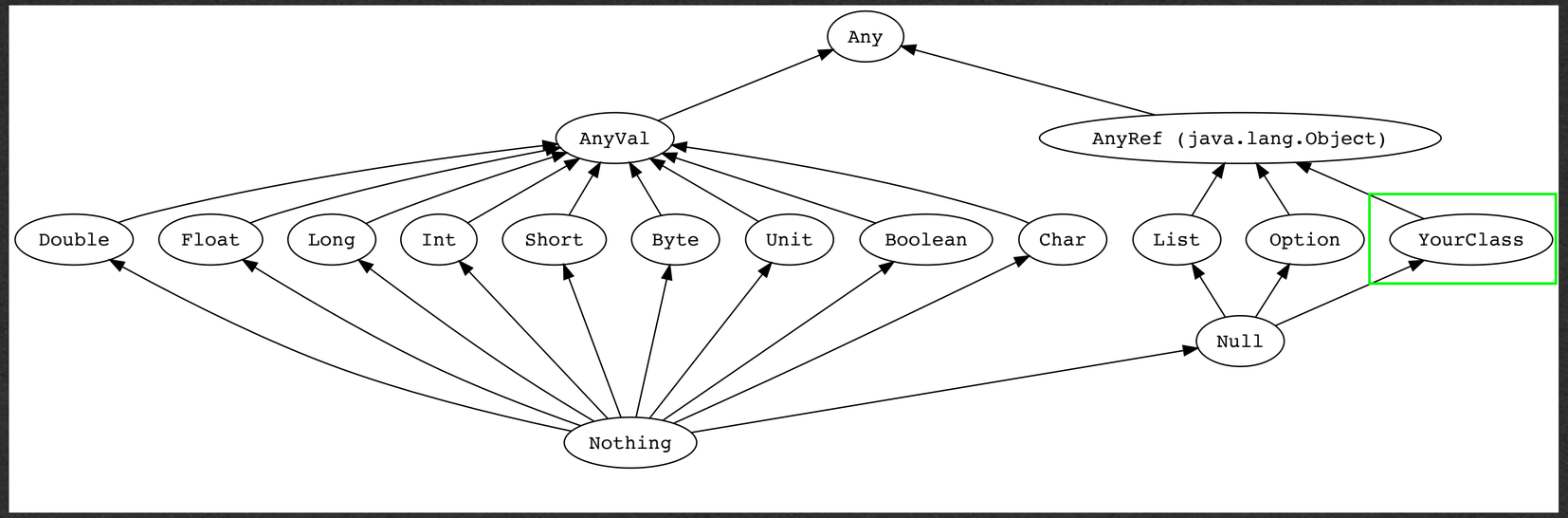
Scala Type Hierarchy



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

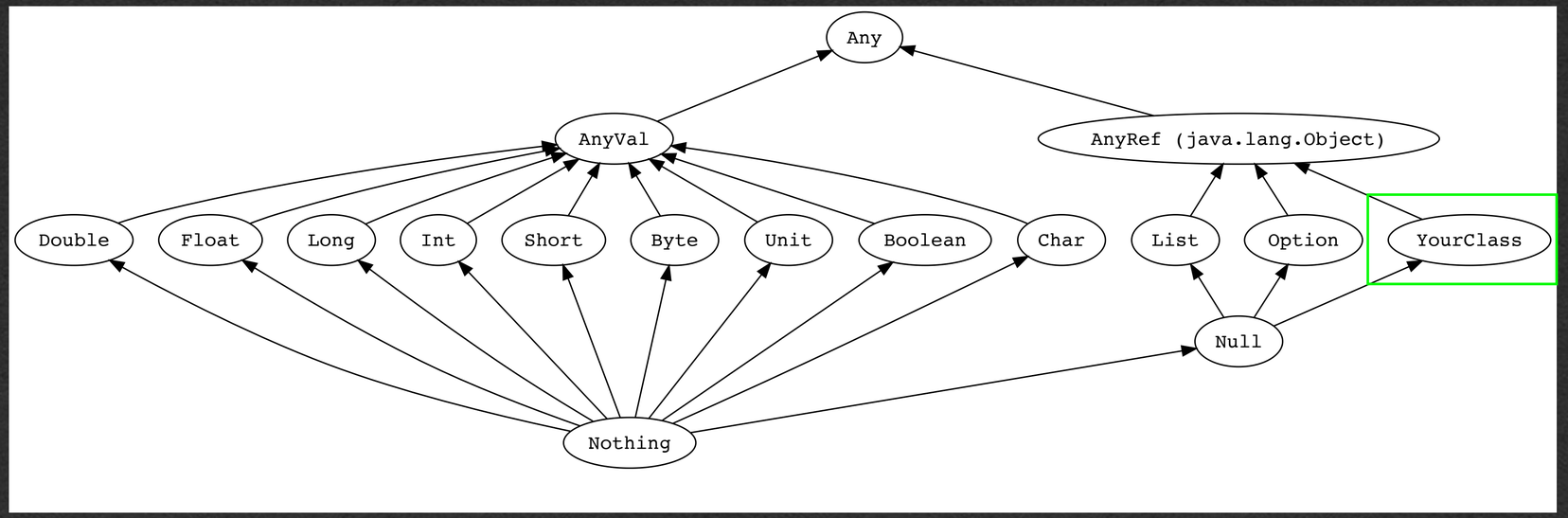
https://docs.scala-lang.org/tour/unified-types.html

Scala Type Hierarchy



- Classes you define extend AnyRef by default
- If you don't explicitly extend any class
  - It's the same as typing "extends AnyRef"

Scala Type Hierarchy



- These two classes are identical

```
abstract class PhysicsObject(var x: Double, var y: Double) {}
```

```
abstract class PhysicsObject(var x: Double, var y: Double) extends AnyRef {}
```

Recall Inheritance

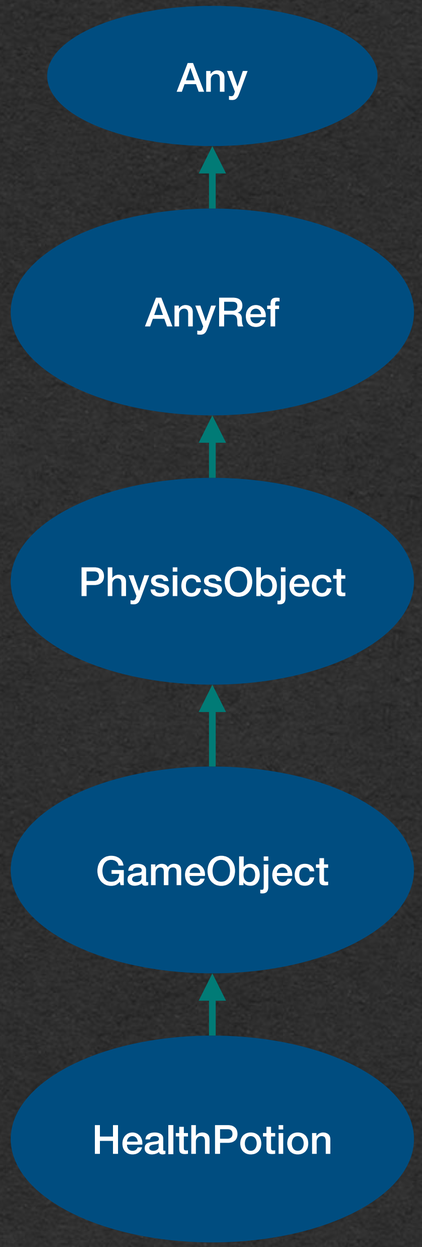
- HealthPotion **extends** GameObject
- GameObject **extends** PhysicsObject
- PhysicsObject **extends** AnyRef
- AnyRef **extends** Any

- HealthPotion **inherits** the **state** and **behavior** of all 5 classes

```
abstract class PhysicsObject(var x: Double, var y: Double) {}
```

```
abstract class GameObject(var xObj: Double, var yObj: Double)
  extends PhysicsObject(xObj, yObj) {

  def objectMass(): Double

  override def toString: String = {
    "(" + this.x + ", " + this.y + "); mass: " + this.objectMass()
  }
}
```

```
class HealthPotion(var xPotion: Double, var yPotion: Double,
                   val volume: Int)
  extends GameObject(xPotion, yPotion) {

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

  override def toString: String = {
    super.toString() + "; volume: " + this.volume
  }
}
```



Recall Inheritance

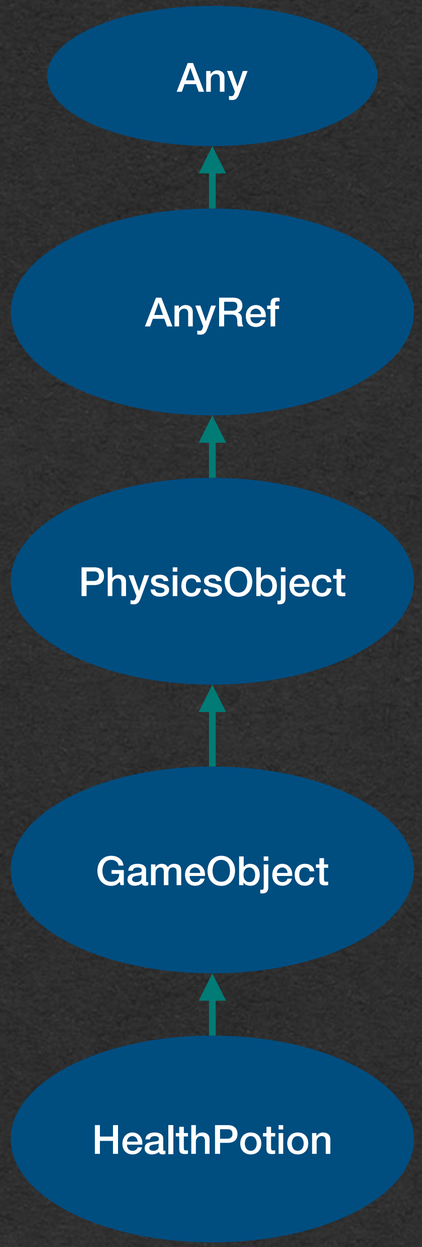
- We call this an "is-a" relationship

- A HealthPotion **is a** GameObject
- A HealthPotion **is a** PhysicsObject
- A HealthPotion **is an** AnyRef
- A HealthPotion **is an** Any

```
abstract class PhysicsObject(var x: Double, var y: Double) {}
```

```
abstract class GameObject(var xObj: Double, var yObj: Double)
  extends PhysicsObject(xObj, yObj) {

  def objectMass(): Double

  override def toString: String = {
    "(" + this.x + ", " + this.y + "); mass: " + this.objectMass()
  }
}
```

```
class HealthPotion(var xPotion: Double, var yPotion: Double,
                   val volume: Int)
  extends GameObject(xPotion, yPotion) {

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

  override def toString: String = {
    super.toString() + "; volume: " + this.volume
  }
}
```



Polymorphism

If an object *is a type*

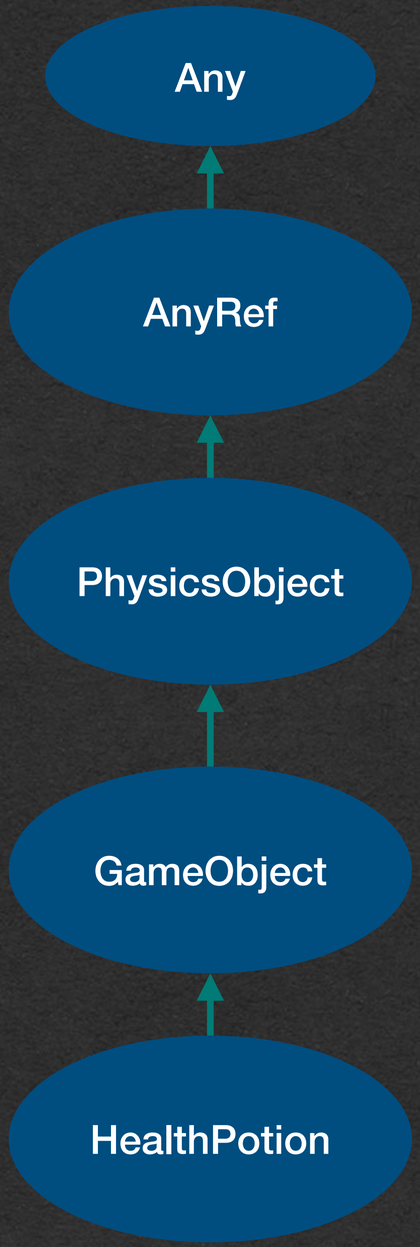
It can be stored in variables of that *type*

Polymorphism

- HealthPotion ***is*** 5 different types

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

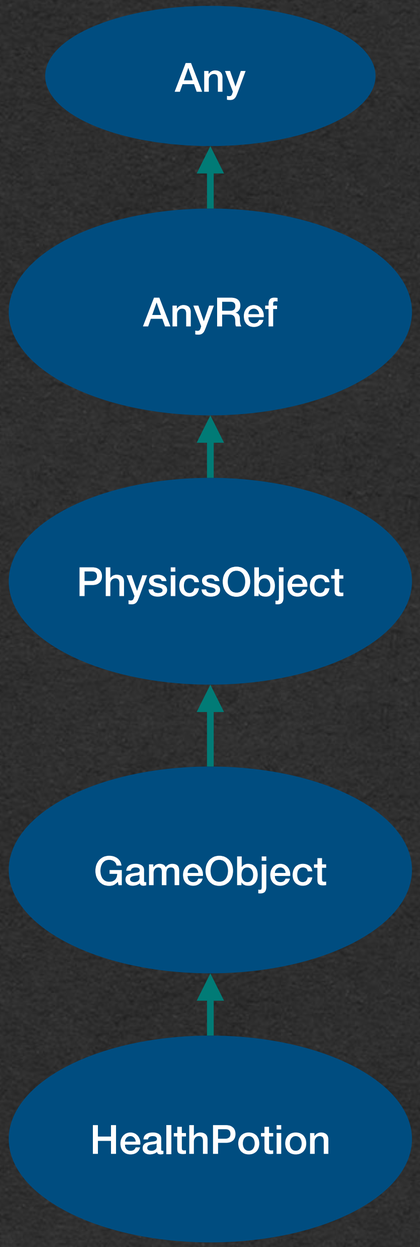
- Can store objects in variables of any of their types



Polymorphism

- All of these assignments are allowed
- HealthPotion has 5 different types!

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



Polymorphism

If an object *is a type*

It can be stored in variables of that *type*

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

Polymorphism

- Can only access state and behavior of the ***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: Any = new HealthPotion(0.0, 0.0, 6)
val potion2: AnyRef = new HealthPotion(0.0, 0.0, 6)
val potion3: PhysicsObject = new HealthPotion(0.0, 0.0, 6)
val potion4: GameObject = new HealthPotion(0.0, 0.0, 6)
val potion5: HealthPotion = new HealthPotion(0.0, 0.0, 6)

potion5.objectMass()
potion4.objectMass()
potion3.objectMass() // Does not compile
```

Polymorphism

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

```
class Player(var x: Double,
             var y: Double,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

  def useBall(ball: DodgeBall): 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 DodgeBall or HeathPotion specific functionality
  - Any state/behavior needed by Player must be in the GameObject class

```
abstract class GameObject(var xObj: Double,
                          var yObj: Double)
  extends PhysicsObject(xObj, yObj) {

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

```
class Player(var x: Double,
             var y: Double,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

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

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

```
class Player(var x: Double,
             var y: Double,
             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 GameObject
- The useItem method will have different behavior depending on the type of its parameter
  - Different implementations of "use" will be called

```
abstract class GameObject(var xObj: Double,
                          var yObj: Double)
  extends PhysicsObject(xObj, yObj) {

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

```
class Player(var x: Double,
             var y: Double,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

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

}
```

```
val ball: DodgeBall = new DodgeBall(0.0, 0.0, 5)
val potion: HealthPotion = new HealthPotion(0.0, 0.0, 5)

val player1: Player = new Player(0.0, 0.0, 20, 12)

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

Polymorphism

- 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(var xObj: Double,
                          var yObj: Double)
  extends PhysicsObject(xObj, yObj) {

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

```
class Player(var x: Double,
             var y: Double,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

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

}
```

```
val ball: DodgeBall = new DodgeBall(0.0, 0.0, 5)
val potion: HealthPotion = new HealthPotion(0.0, 0.0, 5)

val player1: Player = new Player(0.0, 0.0, 20, 12)

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

Polymorphism

- We can also make our player be a GameObject

```
class Player(var x: Double,
             var y: Double,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

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

}
```

```
class Player(var x: Double,
             var y: Double,
             val maxHealth: Int,
             val strength: Int) extends GameObject(x, y) {

  var health: Int = maxHealth

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

  override def objectMass(): Double = {
    this.strength * 20.0
  }

  override def use(player: Player): Unit = {
    player.health = (player.health - this.strength).max(0)
  }
}
```

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(0.0, 0.0, 10, 255)

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

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

PhysicsEngine.doPhysics(gameObjects)
```