Polymorphism

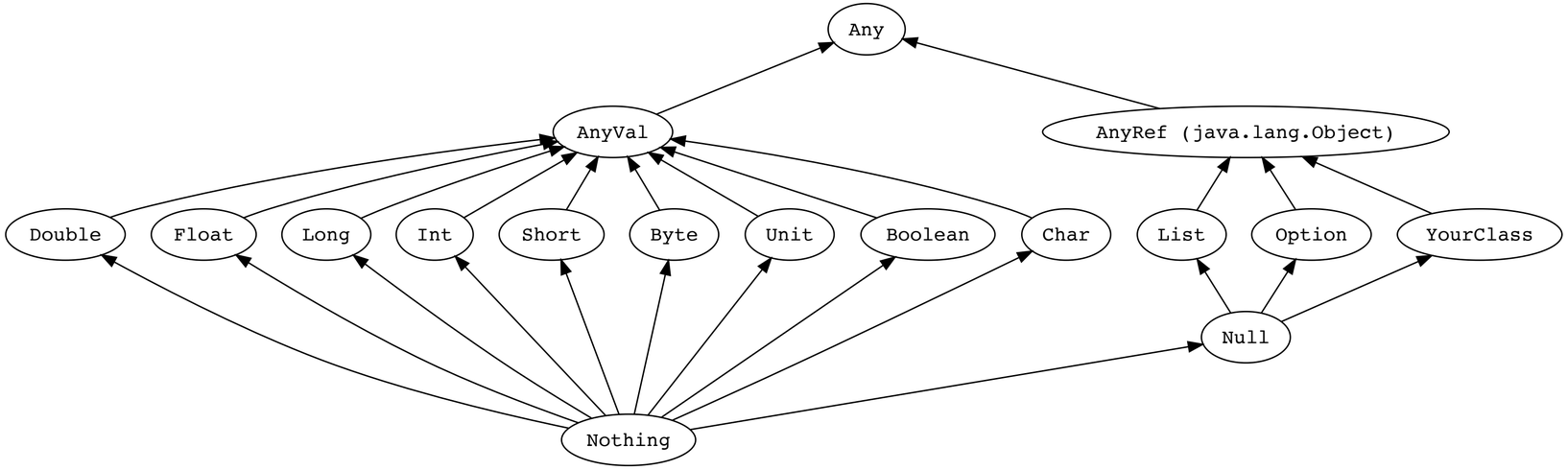
Lecture Question

Question: in a package named "oop.electronics", implement the following to expand the Flashlight/BoomBox functionality from the previous lecture question. Full functionality from the previous lecture question is required

- classes Battery, Electronic, BoomBox, and Flashlight as defined in the previous lecture question
- An **object** named UseElectronics with
  - A method named "useAll" that takes a List of Electronics as a parameter and returns Unit
    - Calls the "use" method on all the Electronics in the input list
    - [Notice that the specific method that is called depends on whether the Electronic is a BoomBox or a Flashlight]
  - A method named "swapBatteries" that takes two Electronics as parameters and returns Unit
    - Exchanges the batteries between the two Electronics

Testing: In a package named "tests" create a Scala class named "TestElectronics" as a test suite that tests all the functionality listed above

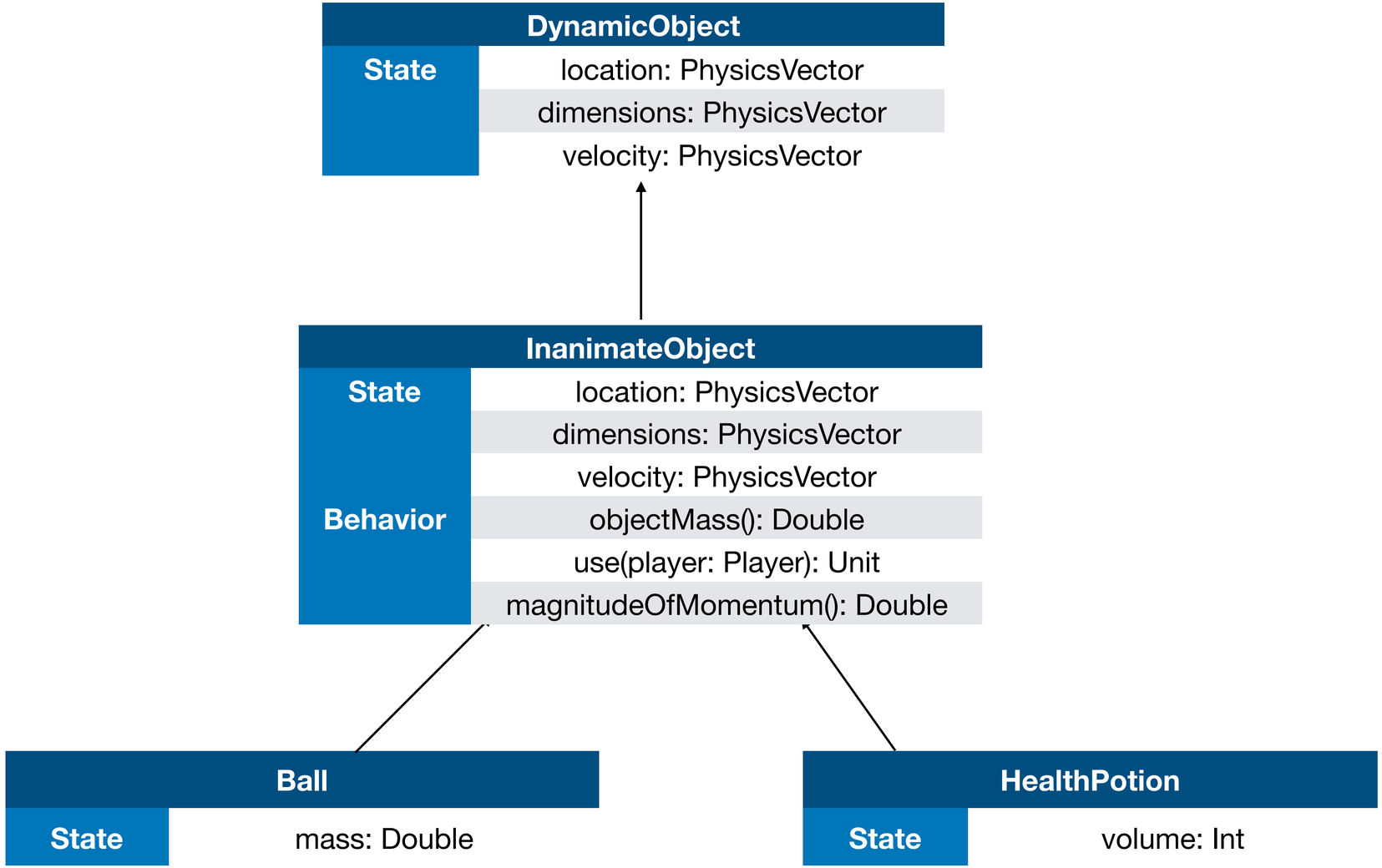
Scala Type Hierarchy



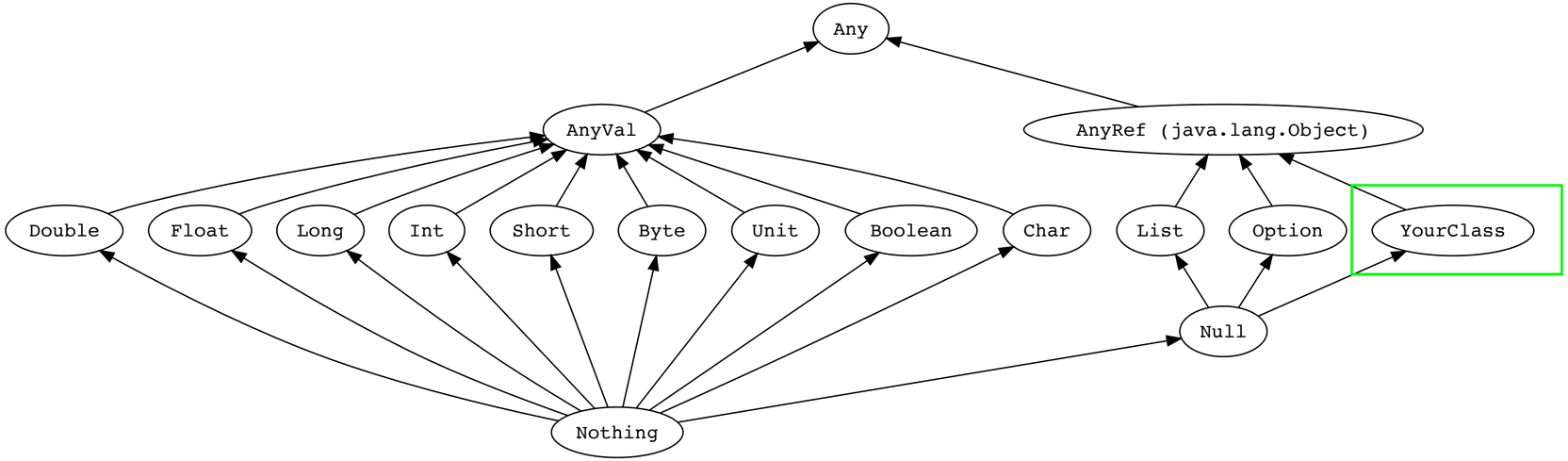
- All objects share Any as their base types
- Classes extending AnyVal will be stored on the **stack**
  - *Unless they are a state variable of an object
- Classes extending AnyRef will be stored on the **heap**

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

Recall


DynamicObject
State
location: PhysicsVector
dimensions: PhysicsVector
velocity: PhysicsVector
InanimateObject
State
location: PhysicsVector
dimensions: PhysicsVector
velocity: PhysicsVector
Behavior
objectMass(): Double
use(player: Player): Unit
magnitudeOfMomentum(): Double
Ball
State
mass: Double
HealthPotion
State
volume: Int

Scala Type Hierarchy



- Classes you define extend AnyRef by default
- HealthPotion has 6 different types

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion2: InanimateObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion3: DynamicObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion4: GameObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion5: AnyRef = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion6: Any = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
```

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

Polymorphism

- HealthPotion has 6 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(), new PhysicsVector(), new PhysicsVector(), 6)
val potion2: InanimateObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion3: DynamicObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion4: GameObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion5: AnyRef = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion6: Any = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
```

Polymorphism

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

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion2: InanimateObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion3: DynamicObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion4: GameObject = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion5: AnyRef = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)
val potion6: Any = new HealthPotion(new PhysicsVector(), new PhysicsVector(), new PhysicsVector(), 6)

potion1.magnitudeOfMomentum()
potion2.magnitudeOfMomentum()
potion3.magnitudeOfMomentum() // 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 dimensions: 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)
  }
}
```

Polymorphism

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

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

  def objectMass(): Double

  def use(player: Player): Unit

}
```

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

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

  var health: Int = maxHealth

  def useItem(item: InanimateObject): 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 InanimateObject(
          location: PhysicsVector,
          velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

}
```

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

  var health: Int = maxHealth

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

}
```

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

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

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

Polymorphism

- We can also make our player be a DynamicObject

```
class Player(var location: PhysicsVector,
             var dimensions: PhysicsVector,
             var velocity: 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,
             _dimensions: PhysicsVector,
             _velocity: PhysicsVector,
             var orientation: PhysicsVector,
             val maxHealth: Int,
             val strength: Int) extends DynamicObject(_location, _dimensions) {

  this.velocity = _velocity
  var health: Int = maxHealth

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

}
```

Polymorphism

- With polymorphism, we can mix types in data structures
  - Something we took for granted in Python/JavaScript
- PhysicsEngine.updateWorld does not care about the types in world.object
  - As long as they all have DynamicObject as a superclass

```
val player: Player = new Player(new PhysicsVector(0.0, 0.0, 0.0),
  new PhysicsVector(1.0, 1.0, 2.0), new PhysicsVector(0.0, 0.0, 0.0),
  new PhysicsVector(1.0, 0.0, 0.0), 10, 255)

val potion1: HealthPotion = new HealthPotion(new PhysicsVector(-8.27, -3.583, 5.3459),
  new PhysicsVector(1.0, 1.0, 1.0), new PhysicsVector(-9.0, 7.17, -9.441), 6)

val potion2: HealthPotion = new HealthPotion(new PhysicsVector(-8.046, -2.128, 5.5179),
  new PhysicsVector(1.0, 1.0, 1.0), new PhysicsVector(6.24, -3.18, -4.021), 6)

val ball1: Ball = new Ball(new PhysicsVector(-2.28, 4.88, 5.1689),
  new PhysicsVector(1.0, 1.0, 1.0), new PhysicsVector(-0.24, 8.59, -6.711), 2)

val ball2: Ball = new Ball(new PhysicsVector(10.325, -2.14, 0.0),
  new PhysicsVector(1.2, 1.2, 1.2), new PhysicsVector(3.65, -9.0, -7.051), 5)

val ball3: Ball = new Ball(new PhysicsVector(-6.988, 1.83, 2.5419),
  new PhysicsVector(1.5, 1.5, 1.5), new PhysicsVector(-3.08, 5.4, 7.019), 10)

val gameObjects: List[DynamicObject] = List(player, potion1, potion2, ball1, ball2, ball3)

val world: World = new World(15)
world.dynamicObjects = gameObjects

PhysicsEngine.updateWorld(world, 0.0167)
```

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

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

```
week4.oop_physics.with_oop.HealthPotion@17c68925
week4.oop_physics.with_oop.HealthPotion@7e0ea639
week4.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,
                   dimensions: PhysicsVector,
                   velocity: PhysicsVector,
                   val volume: Int)
  extends InanimateObject(location, dimensions, velocity) {

...

  override def toString: String = {
    "location: " + this.location + "; velocity: " + this.velocity + "; 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,
                   dimensions: PhysicsVector,
                   velocity: PhysicsVector,
                   val volume: Int)
  extends InanimateObject(location, dimensions, velocity) {

...

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

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

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

Override in Jumper

To create a platform in the jumper game

- Extend JumperObject which extends StaticObject
  - Platforms are now StaticObjects and are compatible with your PhysicsEngine
- Override collideWithDynamicObject to define how an object reacts to a collision with a Platform
  - If the colliding face is the top, the object lands on the Platform

```
class JumperObject(location: PhysicsVector, dimensions: PhysicsVector) extends StaticObject(location, dimensions){
  val objectID: Int = JumperObject.nextID
  JumperObject.nextID += 1
}
```

```
class Platform(location: PhysicsVector, dimensions: PhysicsVector) extends JumperObject(location, dimensions) {

  override def collideWithDynamicObject(otherObject: DynamicObject, face: Integer): Unit = {

    if (face == Face.top) {
      otherObject.velocity.z = 0.0
      otherObject.location.z = this.location.z + this.dimensions.z
      otherObject.onGround()
    }

  }

}
```

Override in Jumper

- Similar method used to create Walls

- Now all dynamic objects in our game react properly to wall and platform collisions as long as they extend DynamicObject

```
class JumperObject(location: PhysicsVector, dimensions: PhysicsVector) extends StaticObject(location, dimensions){
  val objectID: Int = JumperObject.nextID
  JumperObject.nextID += 1
}
```

```
class Wall(location: PhysicsVector, dimensions: PhysicsVector) extends JumperObject(location, dimensions){

  override def collideWithDynamicObject(otherObject: DynamicObject, face: Integer): Unit = {
    if(face == Face.negativeX){
      otherObject.velocity.x = 0.0
      otherObject.location.x = this.location.x - otherObject.dimensions.x
    }else if(face == Face.positiveX){
      otherObject.velocity.x = 0.0
      otherObject.location.x = this.location.x + this.dimensions.x
    }
  }

}
```

Lecture Question

Question: in a package named "oop.electronics", implement the following to expand the Flashlight/BoomBox functionality from the previous lecture question. Full functionality from the previous lecture question is required

- classes Battery, Electronic, BoomBox, and Flashlight as defined in the previous lecture question
- An **object** named UseElectronics with
  - A method named "useAll" that takes a List of Electronics as a parameter and returns Unit
    - Calls the "use" method on all the Electronics in the input list
    - [Notice that the specific method that is called depends on whether the Electronic is a BoomBox or a Flashlight]
  - A method named "swapBatteries" that takes two Electronics as parameters and returns Unit
    - Exchanges the batteries between the two Electronics

Testing: In a package named "tests" create a Scala class named "TestElectronics" as a test suite that tests all the functionality listed above