Inheritance
Question: in a package named "oop.electronics", implement the following. This functionality is the same as the last lecture question, but we will use inheritance to prevent duplicating code.

- class Battery with
  - A constructor that takes a variable named "charge" of type Int

- abstract class Electronic with
  - A constructor that takes no parameters
  - A state variable named "battery" of type Battery
  - A method named "use" that takes no parameters and returns Unit (This can be abstract)
  - A method named "replaceBattery" that takes a Battery as a parameter and returns a Battery
    - This method swaps the input Battery with the Battery currently stored in this Electronic's state variable
    - The returned Battery is the one that was in the state variable when the method is called

- class BoomBox that extends Electronic
  - A constructor that takes a variable of type Battery and assigned it to the inherited state variable named "battery"
    - Your constructor parameter should have a different name than the state variable
  - Override the "use" method to reduce the charge of the battery in the state variable by 3 if its charge is 3 or greater

- class FlashLight that extends Electronic
  - A constructor that takes no parameters
  - When a new FlashLight is created, assign the inherited state variable named "battery" to a new Battery with 5 charge (ie. Batteries included)
  - Override the "use" method to reduce the charge of the battery in the state variable by 1 if its charge is 1 or greater
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
Overview

- Let's do some world building

- If we're making a game, we'll want various objects that will interact with each other

- We'll setup a simple game 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

- Note: We might not build this full game, but we will build some of the game mechanics
Objects Review

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

<table>
<thead>
<tr>
<th>Player</th>
<th>State</th>
<th>location: PhysicsVector</th>
<th>(2.0, -2.0, 2.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dimensions: PhysicsVector</td>
<td>(1.0, 1.0, 2.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>velocity: PhysicsVector</td>
<td>(0.0, -1.0, 0.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>orientation: PhysicsVector</td>
<td>(0.5, -0.5, 0.0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>health: Int</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>maxHealth: Int</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>strength: Int</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Behavior</td>
<td>useBall(ball: Ball): Unit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>useHealthPotion(potion: HealthPotion): Unit</td>
<td></td>
</tr>
</tbody>
</table>

```java
object Player {
    var location: PhysicsVector = new PhysicsVector(2.0, -2.0, 2.0)
    var dimensions: PhysicsVector = new PhysicsVector(1.0, 1.0, 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

<table>
<thead>
<tr>
<th>Ball</th>
<th>State</th>
<th>(1.0, 5.0, 2.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>location: PhysicsVector</td>
<td>(1.0, 1.0, 1.0)</td>
</tr>
<tr>
<td></td>
<td>velocity: PhysicsVector</td>
<td>(1.0, 1.0, 10.0)</td>
</tr>
<tr>
<td></td>
<td>mass: Double</td>
<td>5.0</td>
</tr>
</tbody>
</table>

```scala
object Ball {
  var location: PhysicsVector = new PhysicsVector(1.0, 5.0, 2.0)
  var dimensions: PhysicsVector = new PhysicsVector(1.0, 1.0, 1.0)
  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**

<table>
<thead>
<tr>
<th>HealthPotion</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State</strong></td>
</tr>
<tr>
<td>location: PhysicsVector</td>
</tr>
<tr>
<td>dimensions: PhysicsVector</td>
</tr>
<tr>
<td>velocity: PhysicsVector</td>
</tr>
<tr>
<td>volume: Int</td>
</tr>
<tr>
<td><strong>Behavior</strong></td>
</tr>
<tr>
<td>use(player: Player): Unit</td>
</tr>
</tbody>
</table>

```java
object HealthPotion {
    var location: PhysicsVector = new PhysicsVector(5.0, 7.0, 0.0)
    var dimensions: PhysicsVector = new PhysicsVector(1.0, 1.0, 1.0)
    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)
    }
}
```
- But this is restrictive
- Game can only have one Ball, one HealthPotion, and on Player
- Can play, but not very fun

## Objects Review

<table>
<thead>
<tr>
<th>State</th>
<th>Player</th>
<th>State</th>
<th>Ball</th>
<th>State</th>
<th>HealthPotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>location: PhysicsVector</td>
<td></td>
<td>location: PhysicsVector</td>
<td>(2.0, -2.0, 2.0)</td>
<td>location: PhysicsVector</td>
<td>(5.0, 7.0, 0.0)</td>
</tr>
<tr>
<td>dimensions: PhysicsVector</td>
<td></td>
<td>dimensions: PhysicsVector</td>
<td>(1.0, 1.0, 2.0)</td>
<td>dimensions: PhysicsVector</td>
<td>(1.0, 1.0, 1.0)</td>
</tr>
<tr>
<td>velocity: PhysicsVector</td>
<td></td>
<td>velocity: PhysicsVector</td>
<td>(0.0, -1.0, 0.0)</td>
<td>velocity: PhysicsVector</td>
<td>(0.0, 0.0, 0.0)</td>
</tr>
<tr>
<td>orientation: PhysicsVector</td>
<td></td>
<td>orientation: PhysicsVector</td>
<td>(0.5, -0.5, 0.0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>health: Int</td>
<td></td>
<td>maxHealth: Int</td>
<td>17</td>
<td>volume: Int</td>
<td>3</td>
</tr>
<tr>
<td>maxHealth: Int</td>
<td></td>
<td>strength: Int</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strength: Int</td>
<td></td>
<td>useBall(ball: Ball): Unit</td>
<td></td>
<td>use(player: Player): Unit</td>
<td></td>
</tr>
<tr>
<td>useHealthPotion(potion: HealthPotion): Unit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>useBall(ball: Ball): Unit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>use(player: Player): Unit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Classes Review

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

<table>
<thead>
<tr>
<th>Player</th>
<th>State</th>
<th>Location: PhysicsVector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Dimensions: PhysicsVector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Velocity: PhysicsVector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Orientation: PhysicsVector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Health: Int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MaxHealth: Int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strength: Int</td>
</tr>
<tr>
<td></td>
<td>Behavior</td>
<td>UseBall(ball: Ball): Unit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UseHealthPotion(potion: HealthPotion): Unit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ball</th>
<th>State</th>
<th>Location: PhysicsVector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Dimensions: PhysicsVector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Velocity: PhysicsVector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mass: Double</td>
</tr>
<tr>
<td></td>
<td>Behavior</td>
<td>Use(player: Player): Unit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HealthPotion</th>
<th>State</th>
<th>Location: PhysicsVector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Dimensions: PhysicsVector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Velocity: PhysicsVector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Volume: Int</td>
</tr>
<tr>
<td></td>
<td>Behavior</td>
<td>Use(player: Player): Unit</td>
</tr>
</tbody>
</table>
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)
}
}
# Classes Review

<table>
<thead>
<tr>
<th>Ball</th>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>location: PhysicsVector</td>
<td>use(player: Player): Unit = {</td>
</tr>
<tr>
<td></td>
<td>dimensions: PhysicsVector</td>
<td>this.velocity = new PhysicsVector( player.orientation.x * player.strength, player.orientation.y * player.strength, player.strength ) }</td>
</tr>
<tr>
<td></td>
<td>velocity: PhysicsVector</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>mass: Double</td>
<td>}</td>
</tr>
</tbody>
</table>

```java
class Ball(var location: PhysicsVector,  
var dimensions: 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  
        )
    }
}
```
class HealthPotion(var location: PhysicsVector, 
    var dimensions: PhysicsVector, 
    var velocity: PhysicsVector, 
    val volume: Int) {

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

\begin{tabular}{|c|c|}
\hline
\textbf{HealthPotion} & \\
\hline
\textbf{State} & location: PhysicsVector \\
& dimensions: PhysicsVector \\
& velocity: PhysicsVector \\
& volume: Int \\
\hline
\textbf{Behavior} & use(player: Player): Unit \\
\hline
\end{tabular}
Classes Review

- Use the class to create multiple objects with different states

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

var ball2: Ball = new Ball(
    new PhysicsVector(6.0, -3.0, 2.0),
    new PhysicsVector(1.0, 1.0, 1.0),
    new PhysicsVector(0.0, 4.5, 4.5),
    10.0
) // ball2 stores 21374
```
Inheritance

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

<table>
<thead>
<tr>
<th>State</th>
<th>Ball</th>
<th>State</th>
<th>HealthPotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>location</td>
<td>location: PhysicsVector</td>
<td>location</td>
<td>location: PhysicsVector</td>
</tr>
<tr>
<td>dimensions</td>
<td>dimensions: PhysicsVector</td>
<td>dimensions</td>
<td>dimensions: PhysicsVector</td>
</tr>
<tr>
<td>velocity</td>
<td>velocity: PhysicsVector</td>
<td>velocity</td>
<td>velocity: PhysicsVector</td>
</tr>
<tr>
<td>mass</td>
<td>mass: Double</td>
<td>volume</td>
<td>volume: Int</td>
</tr>
<tr>
<td>Behavior</td>
<td>use(player: Player): Unit</td>
<td>Behavior</td>
<td>use(player: Player): Unit</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Ball</th>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>location: PhysicsVector</td>
<td>use(player: Player): Unit</td>
</tr>
<tr>
<td></td>
<td>dimensions: PhysicsVector</td>
<td></td>
</tr>
<tr>
<td></td>
<td>velocity: PhysicsVector</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mass: Double</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HealthPotion</th>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>location: PhysicsVector</td>
<td>use(player: Player): Unit</td>
</tr>
<tr>
<td></td>
<td>dimensions: PhysicsVector</td>
<td></td>
</tr>
<tr>
<td></td>
<td>velocity: PhysicsVector</td>
<td></td>
</tr>
<tr>
<td></td>
<td>volume: Int</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>location: PhysicsVector</td>
<td></td>
</tr>
<tr>
<td>dimensions: PhysicsVector</td>
<td></td>
</tr>
<tr>
<td>velocity: PhysicsVector</td>
<td></td>
</tr>
<tr>
<td>objectMass(): Double</td>
<td></td>
</tr>
<tr>
<td>use(player: Player): Unit</td>
<td></td>
</tr>
</tbody>
</table>

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

```scala
abstract class InanimateObject(var location: PhysicsVector, var dimensions: PhysicsVector, var 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`

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

  def objectMass(): Double

  def use(player: Player): Unit
}

class Ball(location: PhysicsVector, 
  dimensions: PhysicsVector, 
  velocity: PhysicsVector, 
  mass: Double) extends InanimateObject(location, dimensions, velocity) {

  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
    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

```scala
abstract class InanimateObject(
  var location: PhysicsVector,
  var dimensions: PhysicsVector,
  var velocity: PhysicsVector)
{
  def objectMass(): Double
  def use(player: Player): Unit
}

class Ball(location: PhysicsVector,
           dimensions: PhysicsVector,
           velocity: PhysicsVector,
           mass: Double)
  extends InanimateObject(location, dimensions, velocity) {

  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
    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(
  var location: PhysicsVector,
  var dimensions: PhysicsVector,
  var velocity: PhysicsVector) {

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

class Ball(location: PhysicsVector,
           dimensions: PhysicsVector,
           velocity: PhysicsVector,
           mass: Double)
  extends InanimateObject(location, dimensions, velocity) {

  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
    this.velocity.z = player.strength
  }
}
```
Inheritance

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

```scala
abstract class InanimateObject(
  var location: PhysicsVector,
  var dimensions: PhysicsVector,
  var velocity: PhysicsVector) {
  def objectMass(): Double
  def use(player: Player): Unit
}

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

  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 Ball(location: PhysicsVector,
  dimensions: PhysicsVector,
  velocity: PhysicsVector,
  mass: Double)
  extends InanimateObject(location, dimensions,
  velocity) {

  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
    this.velocity.z = player.strength
  }
}
```
Inheritance

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

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

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Double = {
    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(var location: PhysicsVector, var dimensions: PhysicsVector, var velocity: PhysicsVector) {

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Double = {
    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, dimensions: PhysicsVector, inputVelocity: PhysicsVector) extends DynamicObject(location, dimensions) {

  this.velocity = inputVelocity

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Double = {
    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 DynamicObject!

abstract class InanimateObject(location: PhysicsVector, dimensions: PhysicsVector, inputVelocity: PhysicsVector) extends DynamicObject(location, dimensions) {

  this.velocity = inputVelocity

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Double = {
    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()
  }
}
• Note that the velocity is inherited

• The velocity parameter in the constructor must have a different name that the inherited variable
  • Allows us to assign its value to the state variable
  • They would both be referred to with this causing a name conflict

• No name conflict with multiple location/dimension since they are only in the header

abstract class InanimateObject(location: PhysicsVector, dimensions: PhysicsVector, inputVelocity: PhysicsVector) extends DynamicObject(location, dimensions) {

  this.velocity = inputVelocity

  def objectMass(): Double

  def use(player: Player): Unit

  def magnitudeOfMomentum(): Double = {
    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

DynamicObject

State
- location: PhysicsVector
- dimensions: PhysicsVector
- velocity: PhysicsVector

Behavior
- objectMass(): Double
- use(player: Player): Unit
- magnitudeOfMomentum(): Double

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

- 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

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

```scala
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
Question: in a package named "oop.electronics", implement the following. This functionality is the same as the last lecture question, but we will use inheritance to prevent duplicating code.

- class Battery with
  - A constructor that takes a variable named "charge" of type Int

- abstract class Electronic with
  - A constructor that takes no parameters
  - A state variable named "battery" of type Battery
  - A method named "use" that takes no parameters and returns Unit (This can be abstract)
  - A method named "replaceBattery" that takes a Battery as a parameter and returns a Battery
    - This method swaps the input Battery with the Battery currently stored in this Electronic's state variable
    - The returned Battery is the one that was in the state variable when the method is called

- class BoomBox that extends Electronic
  - A constructor that takes a variable of type Battery and assigned it to the inherited state variable named "battery"
    - Your constructor parameter should have a different name than the state variable
  - Override the "use" method to reduce the charge of the battery in the state variable by 3 if its charge is 3 or greater

- class FlashLight that extends Electronic
  - A constructor that takes no parameters
    - When a new FlashLight is created, assign the inherited state variable named "battery" to a new Battery with 5 charge (ie. Batteries included)
  - Override the "use" method to reduce the charge of the battery in the state variable by 1 if its charge is 1 or greater
This question will be checked with the same tests as the previous question except

The code is in a different package so it doesn't interfere with your code from the previous question. Be sure you check that this import works from a different package in your project.

```scala
import oop.electronics.{Battery, BoomBox, FlashLight, Electronic}
val flashlight1: Electronic = new FlashLight()
val boomBox1: Electronic = new BoomBox(new Battery(10))
```

Your FlashLight and BoomBox classes must inherit Electronic. This will be checked by storing them in variables of type Electronic.

😳 I just noticed that put a capital L in flashlight. I'll leave it for consistency, but I am so sorry if this caused you frustration.