Adapter Pattern

Adapter Pattern

- Is a software design pattern
  - Solution for a specific problem
  - Different from architecture patter (ex. MVC) which organizes the entire project
- Used to make incompatible classes work together

Adapter Pattern

- Need to comply with a specific interface
- Want to use code that complies with a different interface
- Create an adapter class to make them compatible

Want to use

Adapter

Need to comply with

Physics - A Different Perspective

- We want to use an existing 3D library
- The library uses a coordinate system with up as the negative y-axis
  - ScalaFX/JavaFX uses this coordinate system
- Suppose a library with the coordinates revolves around this Object3D trait/interface to know the location of each object

```
trait Object3D {

  /**
    * Object3D is a trait (abstract class with no constructor) to be extended by any objects
    * in a 3D world with a coordinate system the same as in JavaFX/ScalaFX which uses an
    * inverted y axis for the up/down direction meaning that gravity is in the positive y direction
    *
    * This trait defines 3 methods used to get the location of the object. Any object in this world
    * must extend this trait and implement these methods
    */

  def translateX: Double
  def translateY: Double
  def translateZ: Double

}
```

Physics - A Different Perspective

- All objects in this library extend Object3D
- Each type of object implements the 3 location methods
- Since these classes were written for the library they all follow the same coordinate system

```
class GameObject(var x:Double, var y:Double, var z:Double) extends Object3D {

  /**
    * A minimal example of an object that would fit into the coordinate system expected by
    * Object3D.
    */

  override def translateX: Double = this.x
  override def translateY: Double = this.y
  override def translateZ: Double = this.z

  /* GameObject behavior omitted */

}
```

Physics - A Different Perspective

- The library has tons of functionality revolving around the Object3D trait
- We want to use this functionality
- No problem. Just use the library

```
var objectsInGame: List[Object3D] = List()

// Does interesting things with objectsInGame
```

Physics - A Different Perspective

- But what if we want to use our Physics engine with this library?
  - Our physics assigns the positive z-axis to up
  - Our physics revolves around the PhysicalObject class, not Object3D

```
class PhysicalObject(val location: PhysicsVector, val velocity: PhysicsVector) {

}
```

Physics Adapter

- Create an adapted class to use PhysicalObject in the new library
- All physics still applies to our PhysicalObjects
- Send their locations to the other library for added functionality
  - Adapter translates to the other coordinate system

```
import physics.PhysicalObject

class PhysicalObjectAdapter(val adaptedObject: PhysicalObject) extends Object3D {

  override def translateX: Double = adaptedObject.location.x
  override def translateY: Double = -adaptedObject.location.z
  override def translateZ: Double = adaptedObject.location.y

}
```

Physics Adapter

- Store reference to Ball in type PhysicalObject for use with the physics engine
- Wrap the reference in a PhysicalObjectAdapter for use with the new library

```
def main(args: Array[String]): Unit = {

  var objectsInGame: List[Object3D] = List()

  // Create a game object that's intended to work in the coordinate system expected
  // by Object3D
  // This object is 2 units above the ground at position (5, 3) when viewed from overhead
  val naturalGameObject: GameObject = new GameObject(5, -2, 3)

  // can add the object to a list of all object which would work as expected in this
  // coordinate system [Usage of objectsInGame is omitted]
  objectsInGame = naturalGameObject :: objectsInGame

  // Create a Ball as defined earlier in class. This ball is 6 units off the ground at
  // position (2, -4) when viewed from overhead
  val ourObject: PhysicalObject = new Ball(new PhysicsVector(2, -4, 6), new PhysicsVector(0,0,0), 3)

  // Wrap our object in an adapter object to make it work with the other coordinate system
  val ourObjectAdapter: PhysicalObjectAdapter = new PhysicalObjectAdapter(ourObject)

  // Add our object to the game via the adapter
  // Coordinate systems are now compatible
  objectsInGame = ourObjectAdapter :: objectsInGame

}
```

Lecture Question

Pull the Scala examples repo and find the words package for starter code contains:

- AlliterationLibrary (Need to comply with)
  - Uses WordInterface
  - Expects sounds to be all lowercase and in a single comma separated string
- RhymingDictionary (Want to use)
  - Did not use classes. Used List[String] for sounds
  - Sounds were all uppercase and in a list of strings
- Sample usage of each library

Your Task - Write an adapter class named YourWordAdapter that extends WordInterface, takes a word and sounds in its constructor in the format used in RhymingDictionary, and implements the WordInterface methods to comply with the format expected by AlliterationLibrary

* This question will be open until midnight