State Pattern

Lecture Question

No Control Flow allowed for this lecture question! - Same rules as the Calculator/Microwave HW

We will simulate some character behavior in a platforming game where the player can run, duck, and stand still

- In a package named **oop.platformer** write a class named **Player** with the following functionality

Methods:

- duck(): Unit
  - Enters the ducking state. Cannot transition from running to ducking
- standStill(): Unit
  - Enters the standing state
- run(): Unit
  - Enters the running state. Cannot transition from ducking to running
- jumpHeight(): Int
  - Returns 4 if ducking, 3 if standing, and 6 if running
- movementSpeed(): Int
  - Returns 1 if ducking, 5 if standing, and 12 if running

The initial state of the Player is standing

- **You cannot have any control flow is the code you submit. If you have all your lecture questions in a single project, make sure you only submit your code for this LQ**

Design Patterns

- Approaches to common programming design problems
- There are many design patterns
  - We'll only focus on the state pattern in this course
  - For more patterns, search "The Gang of Four"
- The primary goal of design patterns is to simplify the Design and Maintainability of our programs

State Pattern

- Applies Polymorphism

- Every object contains state and behavior
- We use state variables to change the state of an object and its behavior can depend on this state

- **What if we want to significantly change the behavior of an object?**

State Pattern

- **What if we want to significantly change the behavior of an object?**

- Use if statements?
  - if(condition){someBehavior()}
  - else{completelyDifferentBehavior()}

- This will work, but what about maintainability?

State Pattern

- **What if we want to significantly change the behavior of an object?**

- What if we want many different behaviors
  - if(condition){someBehavior()}
  - else if(otherCondition){otherBehavior()}
  - else if(otherCondition){otherBehavior()}
  - else if(otherCondition){otherBehavior()}
  - else{completelyDifferentBehavior()}
- This would all be in a single method
  - Hard to read
  - Hard to maintain
  - Need to re-test existing functionality each time a condition is added

State Pattern

- Let's try using the **state pattern** as an alternative
- Instead of storing each behavior in the same class, we defer functionality to a state **object**
- Have a state variable containing the current state as an **object**
- Change the state as needed
- Decisions made on type (Polymorphism) not value (Conditionals)
- Modularizes code
  - More, but smaller, pieces of functionality
- Easy to add new features without breaking tested features

State Pattern

- State is represented by an abstract class (or trait, interface)
  - Defines the methods that can be called (API)
- Extend the state class for each concrete state
  - One class for each possible state
- Each state will have a reference to the object to which it is attached
  - Use this reference to access other state variables
  - Use this reference to change state

State Pattern - Example

- OK, but what does all that actually mean?
- Let's use the cool-headed Bruce Banner as an example
  - Bruce is a world-class scientist
  - Bruce can successfully drive a car
  - Bruce is not very helpful in a fight

State Pattern - Example

- However.. Make Bruce angry and he'll become The Incredible Hulk!
  - Smashes cars
  - Great in a fight
  - Out of control!

State Pattern - Example

- One man
- Two significantly different behaviors depending on his current state


State Pattern - Example

- To simulate Bruce in a program, we will create one BruceBanner class containing the behavior in both states
- Bruce Banner can use cars and fight very differently depending on his state
- Defer to a State object to determine how he behaves


State Pattern - Example

- To simulate Bruce in a program, we will create one BruceBanner class containing the behavior in both states
- Bruce Banner can use cars and fight very differently depending on his state
- Defer to a State object to determine how he behaves

```
class BruceBanner {

  var state: State = new DrBanner(this)

  def makeAngry(): Unit = {
    this.state.makeAngry()
  }

  def calmDown(): Unit = {
    this.state.calmDown()
  }

  def useCar(car: Car): Unit = {
    this.state.useCar(car)
  }

  def fight(): Unit = {
    this.state.fight()
  }

}
```

State Pattern - Example

- Create State as an abstract class to define all the methods each state must contain (API)
- Extend State for each possible concrete state
- Implement the methods for each state

```
abstract class State(banner: BruceBanner) {

  def makeAngry()

  def calmDown()

  def useCar(car: Car)

  def fight()

}
```

```
class DrBanner(banner: BruceBanner) extends State(banner) {

  override def makeAngry(): Unit = {
    banner.state = new TheHulk(banner)
  }

  override def calmDown(): Unit = {
    println("already calm")
  }

  override def useCar(car: Car): Unit = {
    car.drive(false)
  }

  override def fight(): Unit = {
    println("this won't end well")
  }

}
```

```
class TheHulk(banner: BruceBanner) extends State(banner){

  override def makeAngry(): Unit = {
    println("already angry")
  }

  override def calmDown(): Unit = {
    banner.state = new DrBanner(banner)
  }

  override def useCar(car: Car): Unit = {
    car.smash()
  }

  override def fight(): Unit = {
    println("Hulk Smash!")
  }

}
```

State Pattern - Example

- Since the BruceBanner class stores a variable of type State
  - Don't worry about what actual type state is
  - Through polymorphism, the methods in State must be implemented and can be called
- Pass each new state a reference to BruceBanner
  - Use the keyword **this**
- Since the reference is passed, each state can access Bruce's state variable, including the state itself

```scala
abstract class State(banner: BruceBanner) {

  def makeAngry()

  def calmDown()

  def useCar(car: Car)

  def fight()

}
```

```scala
class BruceBanner {

  var state: State = new DrBanner(this)

  def makeAngry(): Unit = {
    this.state.makeAngry()
  }

  def calmDown(): Unit = {
    this.state.calmDown()
  }

  def useCar(car: Car): Unit = {
    this.state.useCar(car)
  }

  def fight(): Unit = {
    this.state.fight()
  }

}
```

State Pattern - Example

- Having access to the state allows each state to replace itself with a new state
- We call this a state transition

```
abstract class State(banner: BruceBanner) {

  def makeAngry()

  def calmDown()

  def useCar(car: Car)

  def fight()

}
```

```
class DrBanner(banner: BruceBanner) extends State(banner) {

  override def makeAngry(): Unit = {
    banner.state = new TheHulk(banner)
  }

  override def calmDown(): Unit = {
    println("already calm")
  }

  override def useCar(car: Car): Unit = {
    car.drive(false)
  }

  override def fight(): Unit = {
    println("this won't end well")
  }

}
```

```
class TheHulk(banner: BruceBanner) extends State(banner){

  override def makeAngry(): Unit = {
    println("already angry")
  }

  override def calmDown(): Unit = {
    banner.state = new DrBanner(banner)
  }

  override def useCar(car: Car): Unit = {
    car.smash()
  }

  override def fight(): Unit = {
    println("Hulk Smash!")
  }

}
```

State Pattern - Example

- With two states we could have easily used a single conditional and a boolean flag to store the state
    - Arguably simpler than using the state pattern
- The true power of this pattern comes when we have more states

State Pattern - Example

- Meet Professor Hulk
- Bruce Banner transformed as the Hulk with full control
  - Can drive a car **and** is great in a fight

State Pattern - Example

- To add the new state
  - Create a new class and implement the State methods
  - Add a state transition to enter the new state
- **Did not modify any existing functionality!**

```
class ProfessorHulk(banner: BruceBanner) extends State(banner){

  override def makeAngry(): Unit = {
    println("No problem")
  }

  override def calmDown(): Unit = {
    println("Already calm")
  }

  override def useCar(car: Car): Unit = {
    car.drive(true)
  }

  override def fight(): Unit = {
    println("Smash carefully")
  }

}
```

```
class BruceBanner {

  var state: State = new DrBanner(this)

  def makeAngry(): Unit = {
    this.state.makeAngry()
  }

  def calmDown(): Unit = {
    this.state.calmDown()
  }

  def useCar(car: Car): Unit = {
    this.state.useCar(car)
  }

  def fight(): Unit = {
    this.state.fight()
  }

  def learnControl():Unit = {
    this.state = new ProfessorHulk(this)
  }

}
```

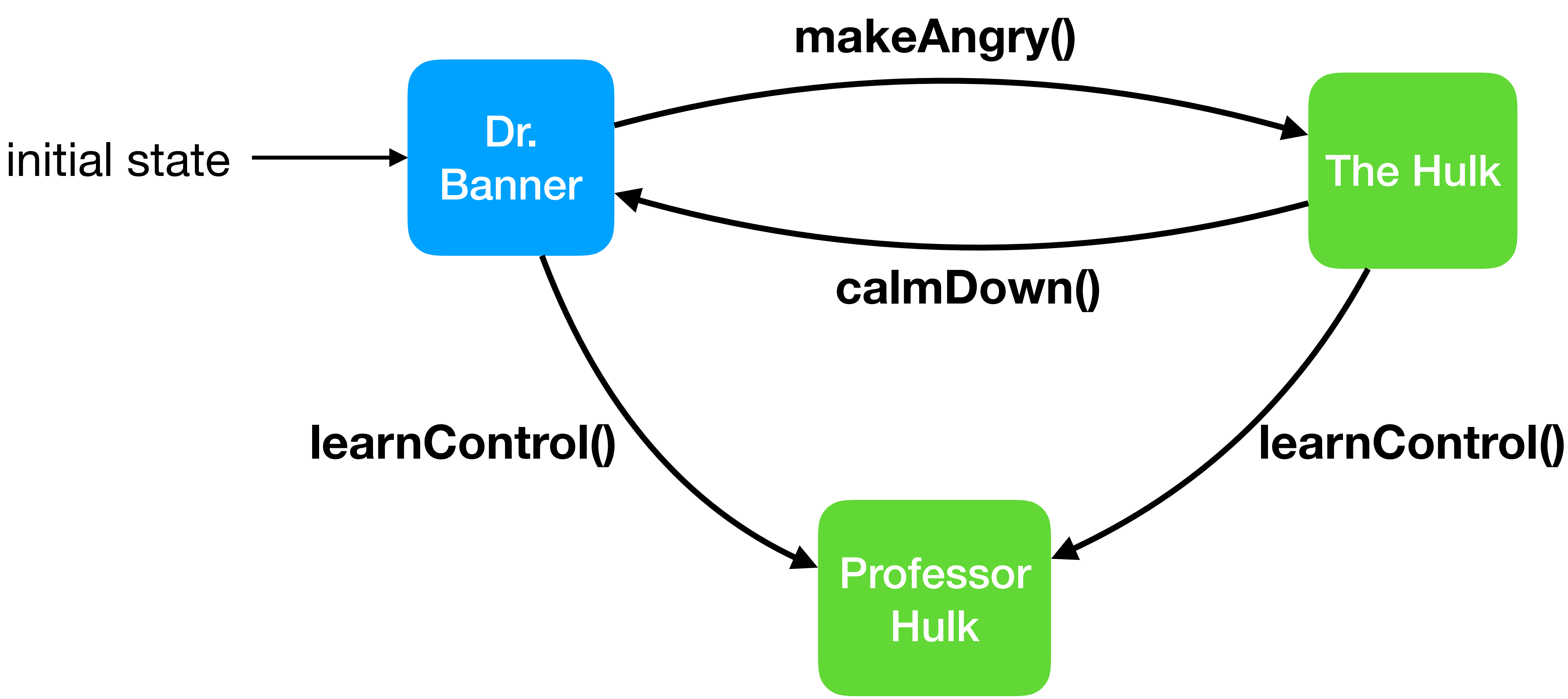
State Pattern - Example

- If we want functionality that is the same in all states
  - Add it to the class containing the state
- Bruce can become Professor Hulk from either of his other states
  - Add this transition to BruceBanner
- Note that there's no going back to the other two states once he becomes Professor Hulk

```
class BruceBanner {

  var state: State = new DrBanner(this)

  def makeAngry(): Unit = {
    this.state.makeAngry()
  }

  def calmDown(): Unit = {
    this.state.calmDown()
  }

  def useCar(car: Car): Unit = {
    this.state.useCar(car)
  }

  def fight(): Unit = {
    this.state.fight()
  }

  def learnControl():Unit = {
    this.state = new ProfessorHulk(this)
  }

}
```

State Pattern - Example

- **State Diagrams**
  - Visualize **states** and **state transitions**
  - Very helpful while designing with the state pattern
- The state diagram for Bruce Banner is as follows



State Pattern - Design

- Write your API
  - What methods will change behavior depending on the current state of the object
  - These methods define your API and are declared in the base state class
- Decide what states should exist
  - Any situation where the behavior is different should be a new state
- Determine the transitions between states

Lecture Question

No Control Flow allowed for this lecture question! - Same rules as the Calculator/Microwave HW

We will simulate some character behavior in a platforming game where the player can run, duck, and stand still

- In a package named **oop.platformer** write a class named **Player** with the following functionality

Methods:

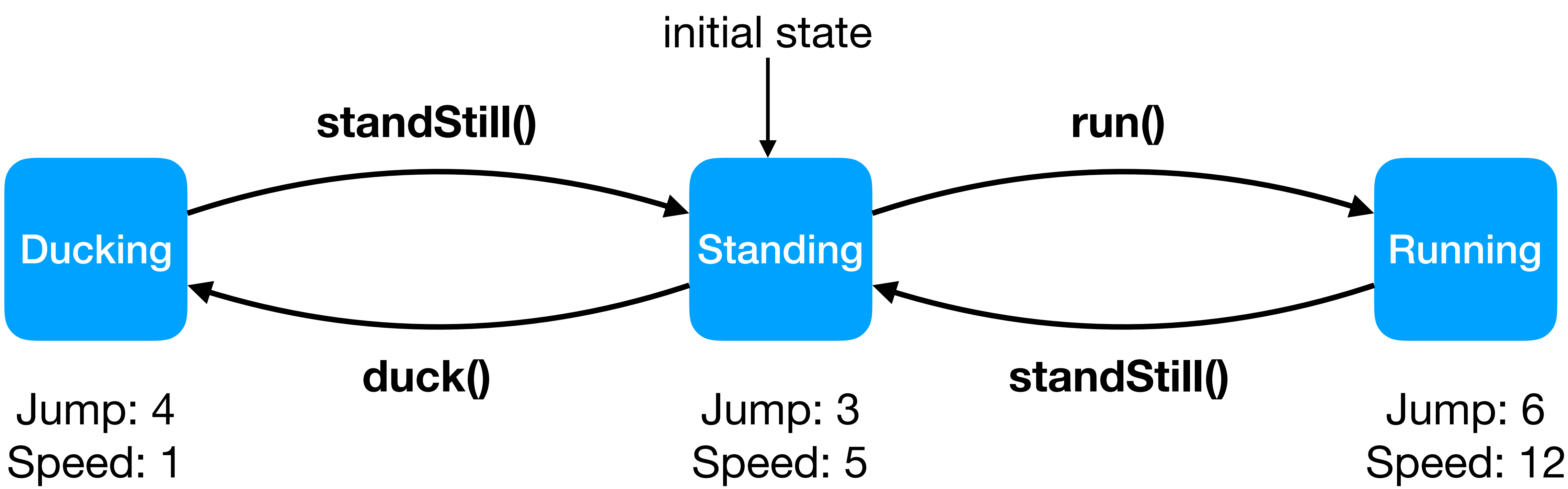
- duck(): Unit
  - Enters the ducking state. Cannot transition from running to ducking
- standStill(): Unit
  - Enters the standing state
- run(): Unit
  - Enters the running state. Cannot transition from ducking to running
- jumpHeight(): Int
  - Returns 4 if ducking, 3 if standing, and 6 if running
- movementSpeed(): Int
  - Returns 1 if ducking, 5 if standing, and 12 if running

The initial state of the Player is standing

- ***You cannot have any control flow is the code you submit. If you have all your lecture questions in a single project, make sure you only submit your code for this LQ**

Lecture Question

To complete this lecture question you are strongly encouraged to use the state pattern. It is possible to use different approaches, though using the state pattern will give you more practice for the HW



Note: You may have to start a new project for this question. Any control flow in your zip file will cause an error even if it's in an unrelated package