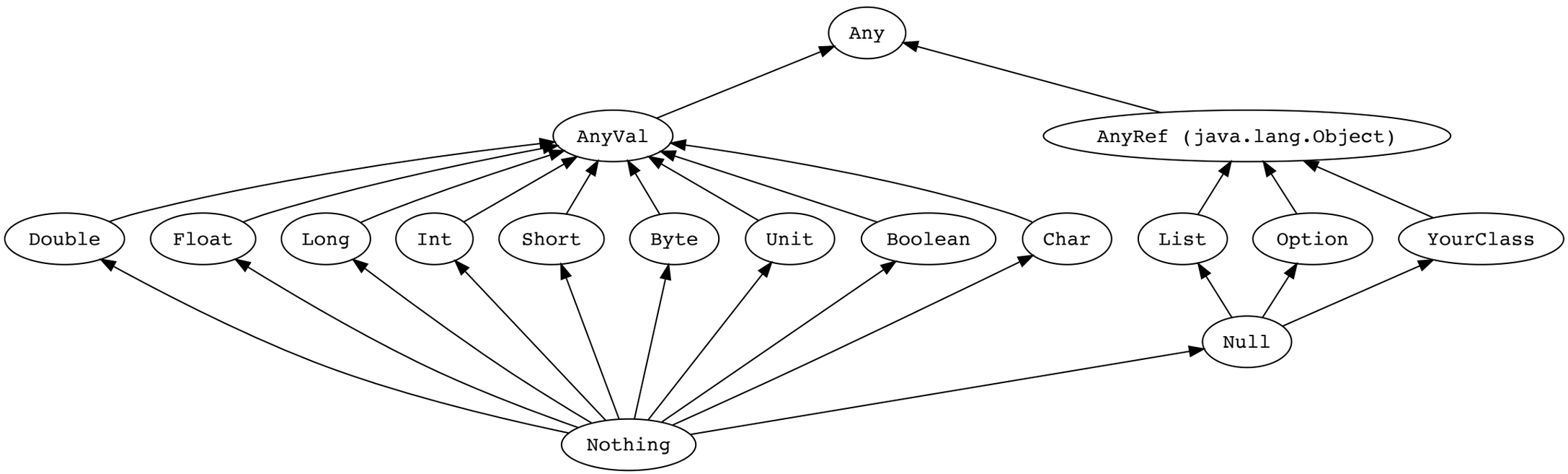
Polymorphism

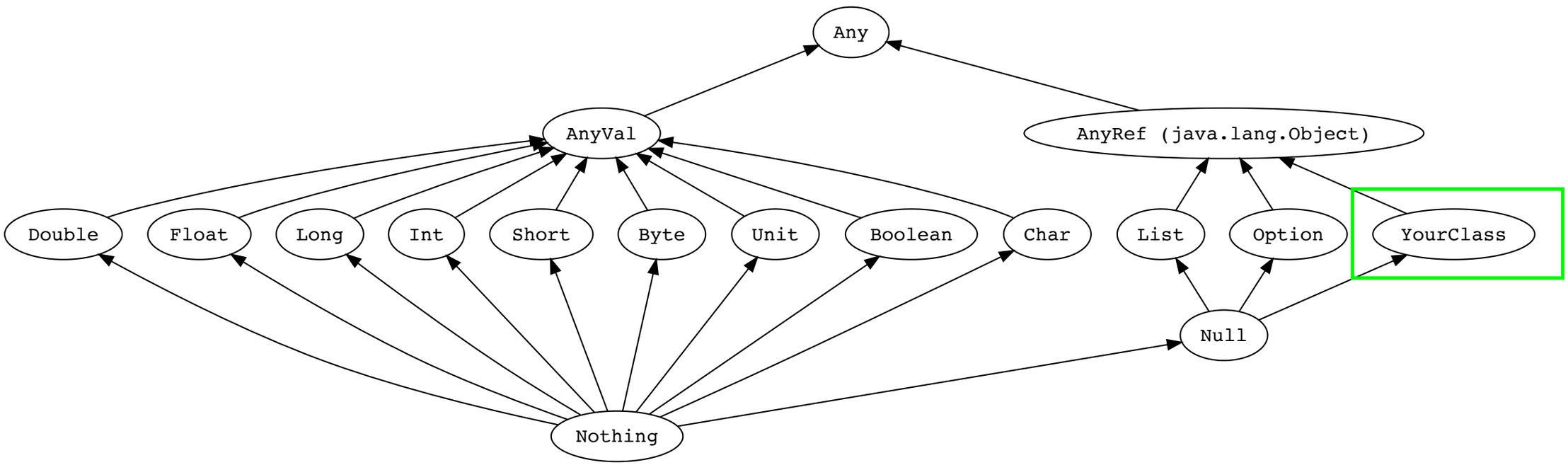
Scala Type Hierarchy



- 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

```
val potion1: HealthPotion = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion2: InanimateObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion3: DynamicObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion4: GameObjectObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion5: AnyRef = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion6: Any = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 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(0,0,0), new PhysicsVector(0,0,0), 6)
val potion2: InanimateObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion3: DynamicObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion4: GameObjectObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion5: AnyRef = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion6: Any = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 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(0,0,0), new PhysicsVector(0,0,0), 6)
val potion2: InanimateObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion3: DynamicObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion4: GameObjectObject = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion5: AnyRef = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)
val potion6: Any = new HealthPotion(new PhysicsVector(0,0,0), new PhysicsVector(0,0,0), 6)

potion1.magnitudeOfMomentum()
potion2.magnitudeOfMomentum()
potion3.magnitudeOfMomentum() // Does not compile
```

Polymorphism

- Why use polymorphism if 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

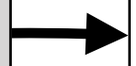
```
class Player(var location: PhysicsVector,
             var dimensions: PhysicsVector,
             var velocity: PhysicsVector,
             var orientation: PhysicsVector,
             val maxHealth: Int,
             val strength: Int) {

  var health: Int = maxHealth

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

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

Polymorphism

- Write function 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 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(potion1, potion2, ball1, ball2, ball3)

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

PhysicsEngine.updateWorld(world, 0.0167)
```

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 a 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
    }
  }

}
```

JSON

An Application of Polymorphism

JSON - Reminder

- JSON is [mostly] used to communicate between programming languages
- Consists of 6 types
  - String
  - Number
  - Boolean
  - Array
  - Object
  - Null

JSON - Reminder

- In Python
  - json.dumps to convert from Python types to JSON string
  - json.loads to convert from JSON string to Python types
- In JavaScript
  - JSON.stringify to convert from JavaScript types to JSON string
  - JSON.parse to convert from JSON string to JavaScript types

JSON

- What about Scala?

{"timestamp":1550774961,"message":"success","iss_position":
{"latitude":"-36.5017","longitude":"-2.8015"}}

- This is valid JSON

- What Scala type do we use to store this data?

JSON

- What about Scala?

{"timestamp":1550774961,"message":"success","iss_position":
{"latitude":"-36.5017","longitude":"-2.8015"}}

- This is valid JSON

- What Scala type do we use to store this data?
  - Map[String, String]?
  - Map[String, Long]?
  - Map[String, Map[String, String]]?
  - Map[String, Any]?? <- This is the only one that can work, but it's very restrictive since we can only use the Any methods

JSON

- What about Scala?

{"timestamp":1550774961,"message":"success","iss_position":
{"latitude":"-36.5017","longitude":"-2.8015"}}

- This is valid JSON

- What Scala type do we use to store this data?
  - We can't mix types in our Scala data structures
  - .. at least, not without polymorphism

JSON - Library

- We'll install a library to help us work with JSON in Scala
  - The Play JSON library
- Library defines these Scala types
  - JsString
  - JsNumber
  - JsBoolean
  - JsArray
  - JsObject
  - JsNull
- **All these types extend JsValue**

JSON - Library

- What about Scala?

{"timestamp":1550774961,"message":"success","iss_position":
{"latitude":"-36.5017","longitude":"-2.8015"}}

- This is valid JSON

- What Scala type do we use to store this data?
  - **Map[String, JsValue]**

JSON - Library

- The library parses JSON strings and converts all values into one of the Js_ types
- We store them in variables of the JsValue base class
- Convert values to the Scala types as needed

Reading JSON

JSON - Library

response = {"timestamp":1550774961,"message":"success","iss_position": {"latitude":"-36.5017","longitude":"-2.8015"}}

```
import play.api.libs.json.{JsValue, Json}

...

val parsed: JsValue = Json.parse(response)

// unused values, but this is how we would extract message and timestamp
val message: String = (parsed \ "message").as[String]
val timestamp: Long = (parsed \ "timestamp").as[Long]
val issLocation: Map[String, String] = (parsed \ "iss_position").as[Map[String, String]]
```

- Use the library to extract specific values

JSON - Library

response = {"timestamp":1550774961,"message":"success","iss_position": {"latitude":"-36.5017","longitude":"-2.8015"}}

```
import play.api.libs.json.{JsValue, Json}

...

val parsed: JsValue = Json.parse(response)

// unused values, but this is how we would extract message and timestamp
val message: String = (parsed \ "message").as[String]
val timestamp: Long = (parsed \ "timestamp").as[Long]
val issLocation: Map[String, String] = (parsed \ "iss_position").as[Map[String, String]]
```

- Import the classes/objects we'll need from the library

JSON - Library

response = {"timestamp":1550774961,"message":"success","iss_position": {"latitude":"-36.5017","longitude":"-2.8015"}}

```
import play.api.libs.json.{JsValue, Json}

...

val parsed: JsValue = Json.parse(response)

// unused values, but this is how we would extract message and timestamp
val message: String = (parsed \ "message").as[String]
val timestamp: Long = (parsed \ "timestamp").as[Long]
val issLocation: Map[String, String] = (parsed \ "iss_position").as[Map[String, String]]
```

- Call Json.parse
- Parses the JSON string and converts it to a JsValue

JSON - Library

response = {"timestamp":1550774961,"message":"success","iss_position": {"latitude":"-36.5017","longitude":"-2.8015"}}

```
import play.api.libs.json.{JsValue, Json}

...

val parsed: JsValue = Json.parse(response)

// unused values, but this is how we would extract message and timestamp
val message: String = (parsed \ "message").as[String]
val timestamp: Long = (parsed \ "timestamp").as[Long]

val issLocation: Map[String, String] = (parsed \ "iss_position").as[Map[String, String]]
```

If the JsValue is an Object:

- Extract values at specific keys
- Use \ to get the value at a key as a JsValue
- Use as[type] to convert the value to the type you expect
    - Cannot use your custom types without defining how to parse your type

Writing JSON

JSON - Library

response = {"timestamp":1550774961,"message":"success","iss_position": {"latitude":"-36.5017","longitude":"-2.8015"}}

```
def createJSON(message: String, timestamp: Long, location: Location): String = {
  val jsonTimestamp: JsValue = Json.toJson(timestamp)
  val jsonMessage: JsValue = Json.toJson(message)

  val locationMap: Map[String, String] = Map(
    "latitude" -> location.latitude.toString,
    "longitude" -> location.longitude.toString
  )

  val jsonLocation: JsValue = Json.toJson(locationMap)

  val jsonMap: Map[String, JsValue] = Map(
    "timestamp" -> jsonTimestamp,
    "message" -> jsonMessage,
    "iss_position" -> jsonLocation
  )

  Json.stringify(Json.toJson(jsonMap))
}
```

- Convert Scala types to JsValue with Json.toJson
  - Cannot use your custom types without defying how to convert your type

JSON - Library

response = {"timestamp":1550774961,"message":"success","iss_position": {"latitude":"-36.5017","longitude":"-2.8015"}}

```
def createJSON(message: String, timestamp: Long, location: Location): String = {
  val jsonTimestamp: JsValue = Json.toJson(timestamp)
  val jsonMessage: JsValue = Json.toJson(message)

  val locationMap: Map[String, String] = Map(
    "latitude" -> location.latitude.toString,
    "longitude" -> location.longitude.toString
  )

  val jsonLocation: JsValue = Json.toJson(locationMap)

  val jsonMap: Map[String, JsValue] = Map(
    "timestamp" -> jsonTimestamp,
    "message" -> jsonMessage,
    "iss_position" -> jsonLocation
  )

  Json.stringify(Json.toJson(jsonMap))
}
```

- Call Json.stringify to convert a type to a JSON string
  - Can be any types known to the library (Most of the common Scala types)

Maven

- We're using a new library
- Must download it before use
- Add it to our Maven file

Maven

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0">
    <groupId>com.mycompany.app</groupId>
    <artifactId>test1</artifactId>
    <modelVersion>4.0.0</modelVersion>
    <url>http://maven.apache.org</url>
    <version>0.0.1</version>

    <dependencies>

        <!-- https://mvnrepository.com/artifact/org.scalatest/scalatest -->
        <dependency>
            <groupId>org.scalatest</groupId>
            <artifactId>scalatest_2.12</artifactId>
            <version>3.0.5</version>
        </dependency>

    </dependencies>

</project>
```

- This is our current Maven file that we used to download scalatest
- We can add more dependancies to this file
    - Open the Maven sidebar, refresh, then download the new libraries

Maven

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0">
    <groupId>com.mycompany.app</groupId>
    <artifactId>test1</artifactId>
    <modelVersion>4.0.0</modelVersion>
    <url>http://maven.apache.org</url>
    <version>0.0.1</version>

    <dependencies>

        <!-- https://mvnrepository.com/artifact/org.scalatest/scalatest -->
        <dependency>
            <groupId>org.scalatest</groupId>
            <artifactId>scalatest_2.12</artifactId>
            <version>3.0.5</version>
        </dependency>

    </dependencies>

</project>
```

- Find new libraries at https://mvnrepository.com
    - An enormous wealth of shared libraries
    - Search for the new libraries, paste the dependency into you pom.xml file
    - Find the play son library and add it to your pom

Lecture Question

Objective: Practice working with JSON strings and Polymorphism in a strongly typed language

Question: In a package named "json" create and complete the "Store" class which is started below

toJSON returns a JSON string representing an object with keys "cashInRegister" and "inventory" mapping to values from the two state variables of the same names

fromJSON takes a JSON string in the same format returned from toJSON and sets the state variables to the values from the JSON string

```
package json

class Store(var cashInRegister: Double, var inventory: List[String]) {

  def toJSON(): String = {
    """
  }

  def fromJSON(jsonString: String): Unit = {

  }

}
```