Oefeningen Lua Scripting

Doel

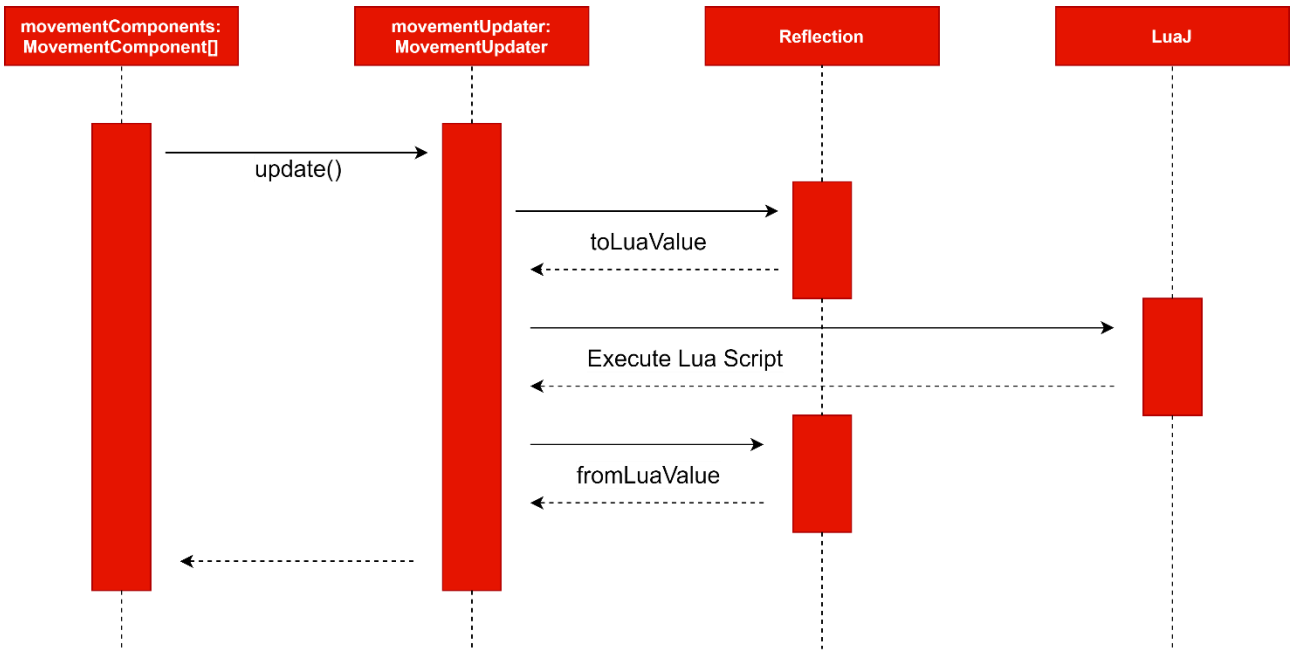
In deze opgave zal je gebruik maken van een scripting taal om dynamisch gedrag toe te voegen aan je applicatie zonder dat je ze moet hercompileren. Dit is vooral nuttig naarmate je applicatie groter wordt en het compileren ervan steeds langer duurt. Door een deel van je applicatie logica uit je eigenlijke applicatie te halen en in script files te steken kan je voorkomen dat je je applicatie moet hercompileren als het gedrag van je applicatie verandert.

Games zijn hier een fantastisch voorbeeld van. In games wordt een deel van de game logica (zoals het gedrag van NPCs) vaak uit de game-code gehaald en in script files gestoken. Op deze manier kunnen mensen die weinig van programmeren kennen toch wijzigingen maken aan het gedrag van entiteiten in het spel zonder het spel zelf te moeten hercompileren.

De scripting taal die wij zullen gebruiken heet Lua. Lua is erg populair onder game developers en was vroeger ook enorm belangrijk voor machine learning (De torch library is origineel in Lua geschreven).

In dit labo zullen we gebruik maken van de Lua scripting taal en de LuaJ library, om enkele functies te bouwen die eender welk object kunnen omzetten van een Java klasse naar een Lua table en vice versa. Hierna zullen we deze functies gebruiken om een movement systeem te bouwen dat we kunnen gebruiken in games.

In de figuur hieronder vind je een overzicht van alles dat we in dit labo zullen proberen maken. Dit diagram is een sequentiediagram. Tijd loopt van boven naar beneden, en interactie tussen componenten wordt voorgesteld met horizontale lijnen. Verticale balken stellen perioden van activiteit voor.



Opgave

Om te zorgen voor interoperabiliteit tussen Java en Lua zullen we gebruik maken van de LuaJ library.

Fibonacci

De eerst opgave is vrij eenvoudig, je zal een klein beetje Java code gebruiken dat gebruik maakt van LuaJ om enkele getallen uit de Fibonacci reeks uit te rekenen. Zo leer je werken met de LuaJ library, die we later in dit labo zullen gebruiken om een complexer systeem te bouwen.

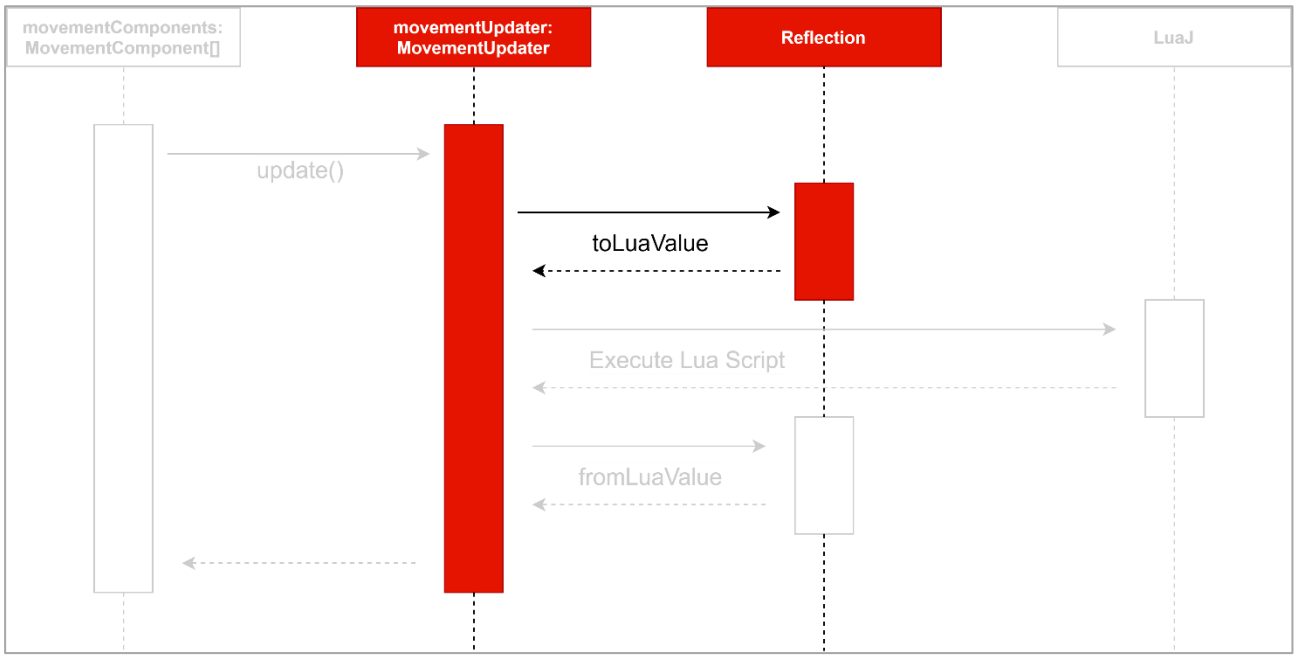
De Lua code hieronder berekent het n-de getal uit de Fibonacci reeks. Je kan dit bestand opslaan als “fib.lua” en de LuaJ library gebruiken om de functie aan te roepen.

```
function fib(n)
    v_n = 0
    v_n_1 = 1

    for i=0,n do
        new = v_n + v_n_1
        v_n = v_n_1
        v_n_1 = new
    end

    return v_n
end
```

Omdat Lua een geïnterpreteerde taal is (Net als Python, maar in tegenstelling tot Java en C++) zal je eerst heel de Lua file moeten doorlopen, de “fib” functie hieruit moeten halen, een LuaValue moeten maken voor n, de “fib” functie moeten aanroepen met n, en het resultaat bijhouden in een nieuwe LuaValue. Tenslotte kan je de LuaValue met het resultaat terug omzetten naar een Java int, en het resultaat weergeven.

Reflection en Serialization

Nu zullen we alles iets complexer maken. In het vorige voorbeeld heb je gezien dat we gebruik maken van LuaValues om data van en naar Lua door te geven. Je volgende taak is om een statische methode te schrijven die een Java Object (type "Object") omzet naar een LuaValue. Specifiek zal je een Lua Table aanmaken, waar de keys overeenkomen met de namen van de velden, en de values overeenkomen met de waarden van de velden. Later kunnen we deze functie gebruiken om eender welk type object te delen tussen Java en Lua. Het schema hieronder toont hoe deze functie in het volledige labo past.



De code hieronder vormt de interface die je zal moeten invullen.

```
public class Reflection
{
	public  static LuaValue toLuaValue(Object object)
	{
		// Fill this in...
	}
}
```

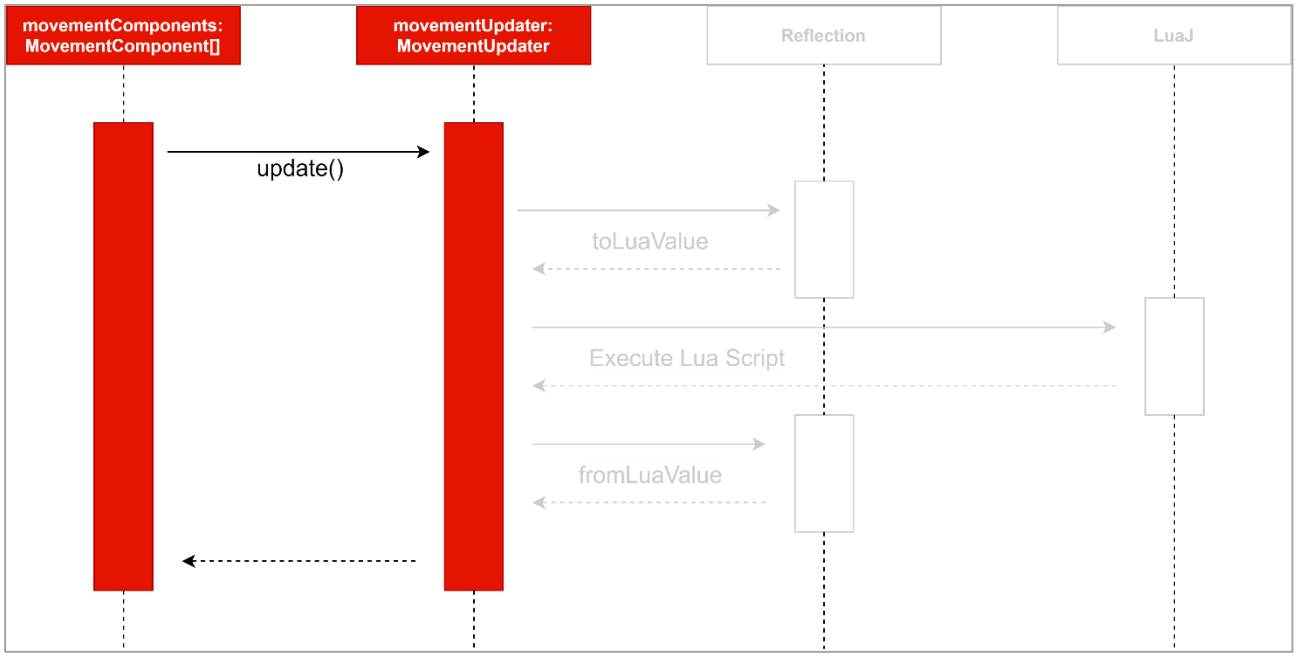
Om deze functionaliteit te realiseren, zullen we eerst gebruik maken van de Java "Class" klasse. De Class klasse is een klasse die klassen kan voorstellen. We kunnen objecten van het Class type gebruiken om te zien welke velden, methoden, constructors, … een klasse heeft. We kunnen zelfs een stapje verder gaan, en at runtime nieuwe klassen maken. Eerst zullen we dus alle velden van het gekregen object moeten zoeken, zodat we ze later in een Lua Table kunnen plaatsen. Dit soort technieken (Code die andere code inspecteert) wordt "reflectie" (Reflection) genoemd.

Om een Lua Table te maken kan je gebruik maken van de Field klasse (die je kan ophalen uit een Class) om de naam en de waarde van elk veld in een object te vinden. Voorlopig moet je alleen velden van het "float" type ondersteunen, maar je kan later ook ondersteuning toevoegen voor andere types (Kan je velden ondersteunen die een klasse als type hebben?). Je zal dus eerst een lijst met alle velden moeten maken, en hieruit de velden filteren die als type "float" hebben.

Nu we alleen nog de velden overhebben met als type float zullen we een Lua table opbouwen. De keys in deze Lua table zijn de namen van de velden die we hebben gevonden, terwijl de values in deze lua table de waarden zijn van deze velden.

Movement Component – Part 1

Om deze methode uit te proberen, zullen we een klein data-oriented movement systeem te bouwen. Hieronder tonen we hoe deze klassen en methoden in het geheel passen:



De eerste stap is het maken van een MovementComponent, die ziet er zo uit:

```
public class MovementComponent
{
    // Velocity
    public float vx;
    public float vy;

    // Acceleration
    public float ax;
    public float ay;
}
```

Je kan deze Velden ook private maken als je dat wilt, maar daar komt wat extra werk bij kijken.

Zorg dat je ook zeker een constructor toevoegt die geen argumenten neemt, je zal dit later nodig hebben!

Zoals je kan zien is onze MovementComponent heel eenvoudig. De klasse bevat een snelheid (vx en vy) en een acceleratie (ax en ay). We zullen de logica voor het updaten van deze klasse ook eenvoudig houden. Als we een MovementComponent updaten zullen we de acceleratie bij de snelheid optellen (v += a) en zullen we de acceleratie met 0.1 vermenigvuldigen (a *= 0.1).

Je mag deze logica nu in een Lua script implementeren. Je Lua script bevat 1 functie “update”, deze functie neemt 1 argument “components”, dit is een lijst van tables, die overeenkomen met onze MovementComponent klasse. Op dit moment moet de “update” functie nog niets returnen.

De volgende stap is ons “system” (in DoD termen), voor ons zal de MovementUpdater deze rol vervullen:

```
public class MovementUpdater
{
        public MovementComponent[] update(final MovementComponent[] components)
        {
                // Fill this in ...
        }
}
```

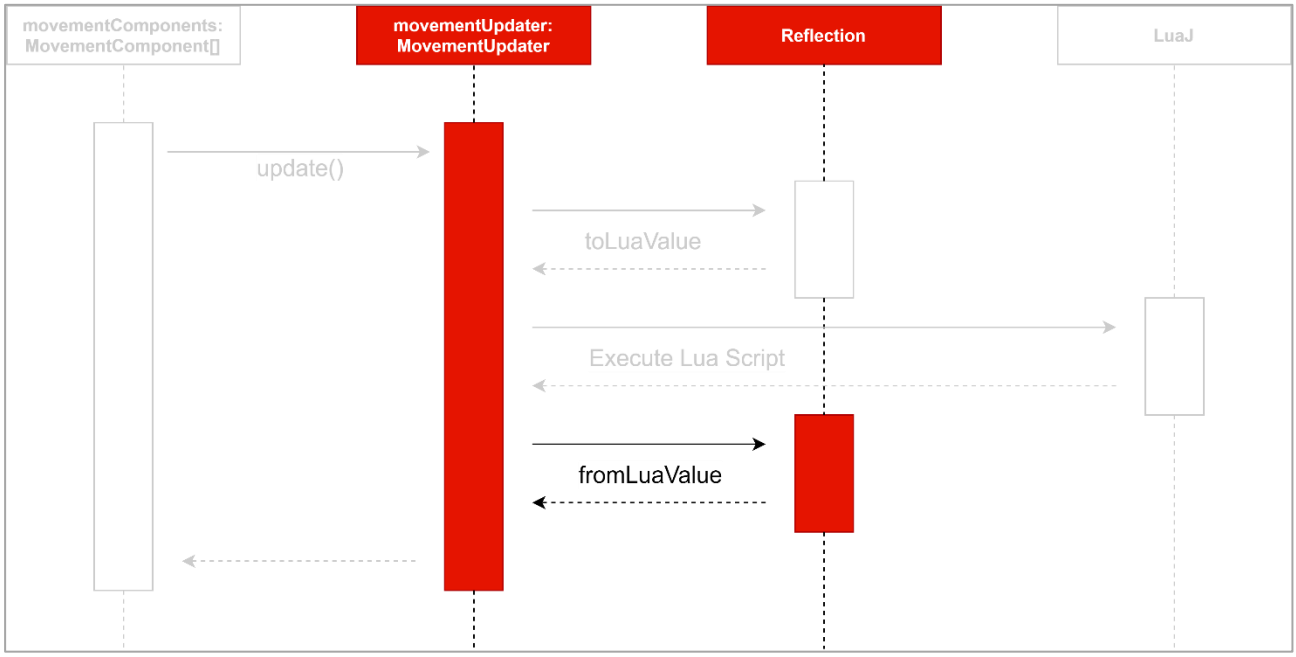
Het is aan jou om de update() methode aan te vullen. Initieel mag je gewoon de input array returnen, we zullen in de volgende stap gebruik maken van Lua om een output te genereren.

Voor de update() methode moet dit het volgende doen:

1. De array van MovementComponents moet worden omgezet naar een array van LuaValues, met de statische methode die je aan het begin van deze stap schreef.
2. Hierna zullen we de array van LuaValues omzetten naar 1 grote LuaValue. Deze grote LuaValue is gewoon een Lua Table met integer keys die de indices voorstellen, en de individuele LuaValues als values. Om te zien hoe je dit best doet kan je best eens een kijkje nemen naar de listOf methode.
3. Tenslotte zal je de update() functie in je Lua script aanroepen, de array-LuaValue uit de vorige stap mag je meegeven als argument. Als test kan je aan het einde van je LuaScript even over al je MovementComponents lopen en hun inhoud printen, zodat je zeker bent dat alles correct werkt.

Reflection en Deserialization

In de vorige stap hebben we de conversie van Java → Lua geïmplementeerd, in deze stap zullen we de omgekeerde stap implementeren, zodat we ook data kunnen terug krijgen uit onze Lua code. Ook deze stap is zichtbaar in ons sequentiediagram:



We zullen beginnen door een nieuwe statische methode te schrijven, gelijkaardig aan die uit de vorige stap:

```
public class Reflection
{
    public static LuaValue toLuaValue(Object object)
    {
        // ...
    }

    public static Object fromLuaValue(LuaValue lua, Class classObj)
    {
        // ...
    }
}
```

We zullen deze methode een LuaValue geven waar we data uit willen halen, en het Class object dat aangeeft welke klasse we verwachten.

Om een object te maken van een bepaalde Class zullen we eerst de Class moeten gebruiken om een geschikte Constructor te vinden, en vervolgens aan te roepen. Gelukkig hebben we een constructor voorzien die geen argumenten neemt, dit maakt alles voor ons veel eenvoudiger. Je kan nu het Class object gebruiken om alle constructoren te vinden, en de constructor te selecteren die geen argumenten neemt.

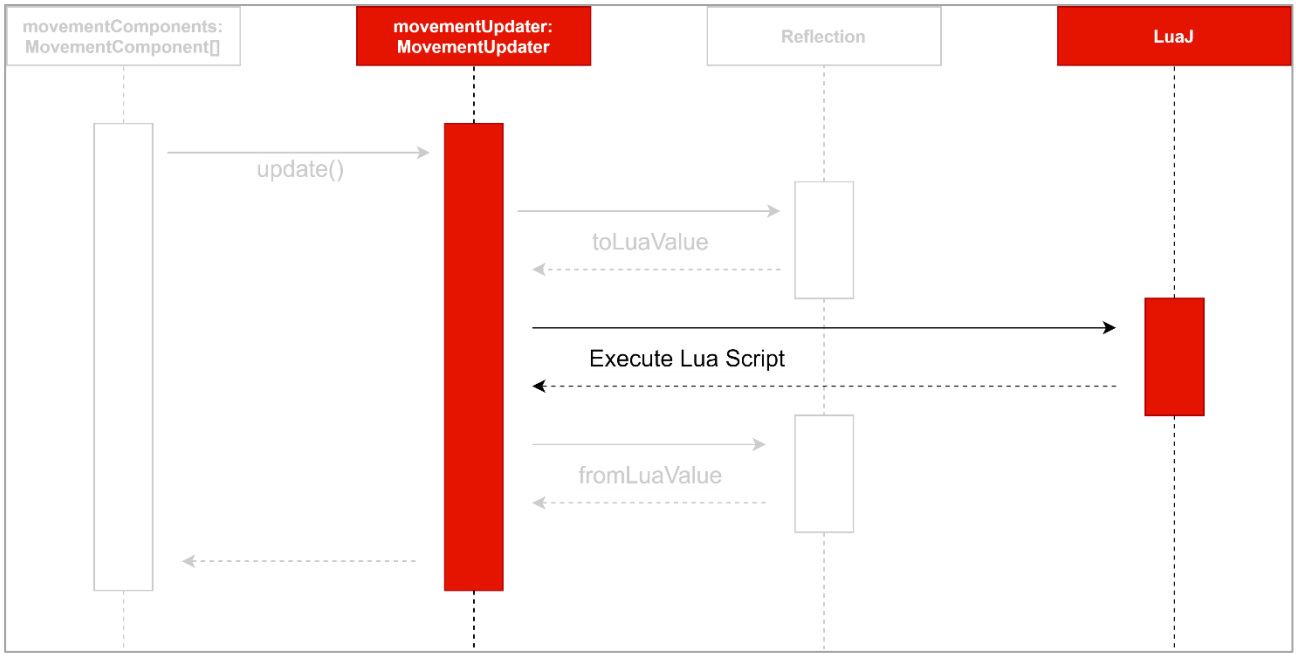
Nu we een geschikte Constructor gevonden hebben, kan je deze gebruiken om een nieuwe Object aan te maken.

Nu we ons object hebben, kunnen we beginnen met het toekennen van de waarden van elk veld. Je kan het Class object gebruiken om alle Fields te vinden, en dan kan je met elk Field, en je nieuw Object de waarden van deze Velden instellen door ze uit de LuaValue te halen. Normaal gezien is deze LuaValue een Table, en kan je de waarden gewoon ophalen aan de hand van de naam van het Field.

Nu alle velden de correcte waarde hebben, kunnen we ons object returnen, naar het correcte type casten en het object beginnen gebruiken!

Movement Component – Part 2

Met deze methode kunnen we onze MovementUpdater afmaken. De eerste wijziging die je mag maken is dat we vanaf nu de ge-update MovementComponents zullen returnen vanuit onze Lua functie.



Uiteraard moeten we deze nieuwe componenten dan ook opslaan in onze movement updater. Herinner je dat een array in Lua gewoon een Table is met integer keys. Vergeet ook niet dat je de lengte van een Lua array kan bepalen door te kijken wanneer je een “nil” waarde tegenkomt. Je kan dan de MovementComponents 1 voor 1 uit de LuaValue halen die onze functie returnt, en a.d.h.v. de statische methode die we eerder schreven terug omzetten naar MovementComponent objecten.

Nu je een lijst hebt met MovementComponents kan je deze returnen, zodat je de input array niet meer moet returnen.

Conclusie

We hebben nu een scriptable movement-systeem opgezet. Als we besluiten om extra velden aan onze MovementComponent toe te voegen, zullen die ook automatisch meegegeven worden aan onze Lua code, en als we besluiten om het movement gedrag te wijzigen, dan moeten we gewoon het Lua script aanpassen, zonder dat we de applicatie zelf moeten wijzigen.