Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
Linguagens e Ambientes de Execução
2022

A biblioteca Java `jsonaif` oferece processamento de dados em formato JSON (https://www.json.org/). Esta biblioteca disponibiliza um objecto `JsonParserReflect` que pode ser usado para transformar uma *string* JSON numa instância de uma classe de domínio compatível (e.g. `Student`) conforme ilustrado no exemplo seguinte:

```
val json = "{ name: \"Ze Manel\", nr: 7353}"
val student = JsonParserReflect.parse(json, Student::class) as Student
assertEquals("Ze Manel", student.name)
assertEquals(7353, student.nr)
```

A class `JsonParserReflect` usa uma instância de uma classe auxiliar `JsonTokens` para percorrer os elementos da *String* JSON fonte. O algoritmo de `JsonParserReflect` é recursivo, criando instâncias de classes de domínio, ou uma lista, e preenchendo os seus campos, ou elementos, com valores primitivos ou instâncias de outras classes de domínio, ou listas, e assim sucessivamente.

A implementação de `JsonParserReflect` mantem uma estrutura de dados com instâncias de `Setter` para cada classe de domínio, de modo a que não seja repetido o trabalho de leitura de metadata via Reflexão. Por exemplo, no parsing de um array de `Student` as propriedades a serem afectadas só devem ser procuradas 1 vez.
A interface `Setter` especifica a forma de afectação de uma determinada propriedade no parâmetro `target` a partir do valor obtido do parâmetro `tokens`:

```
interface Setter {
    fun apply(target: Any, tokens: JsonTokens)
}
```

Por exemplo, na estrutura de dados seguinte cada classe de domínio é mapeada num conjunto de pares: nome da propriedade - `Setter`

```
val setters = mutableMapOf<KClass<*>, Map<String, Setter>>()
```

A classe`JsonParserReflect` suporta duas formas de instanciar a classe de domínio:

1. Através da chamada ao construtor sem parâmetros, ou que tem todos os parâmetros opcionais (https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.reflect.full/create-instance.html). E.g. `Student`
2. Chamando um construtor com parâmetros. E.g. `Person`

A implementação de `parseObject` dá prioridade à opção 1, sempre que possível.

As propriedades da classe de domínio podem ter nomes distintos dos nomes usados na representação em JSON. Por exemplo, uma propriedade em JSON pode ter o nome `birth_date` e em Kotlin `birthDate`. Para resolver a correspondência entre propriedades de nome distinto a anotação `JsonProperty` pode ser usada sobre propriedades de uma classe de domínio indicando o nome correspondente em JSON (e.g. `@JsonProperty(“birth_date”)`).

Existe uma forma alternativa de definir o valor de objectos sem ter que seguir a sintaxe JSON. Por exemplo, em vez de a propriedade `birth` de `Person`, do tipo `pt.isel.sample.Date`, ser definida em JSON, como no exemplo seguinte, poderá ter uma forma alternativa como a que se apresenta para `Student`:

- JSON for a Person: `"{ name: "Ze Manel", birth: { year: 1999, month: 9, day: 19}, sibling: { name: "Kata Badala"}}"`
- JSON for a Student: `"{ name: "Maria Papoila", nr: 73753, birth: "1998-11-17" }"`

Neste caso a propriedade `birth` em `Student` tem que ter uma anotação que identifique a classe responsável por fazer a conversão de `String` numa instância de `Date`. Exemplo:

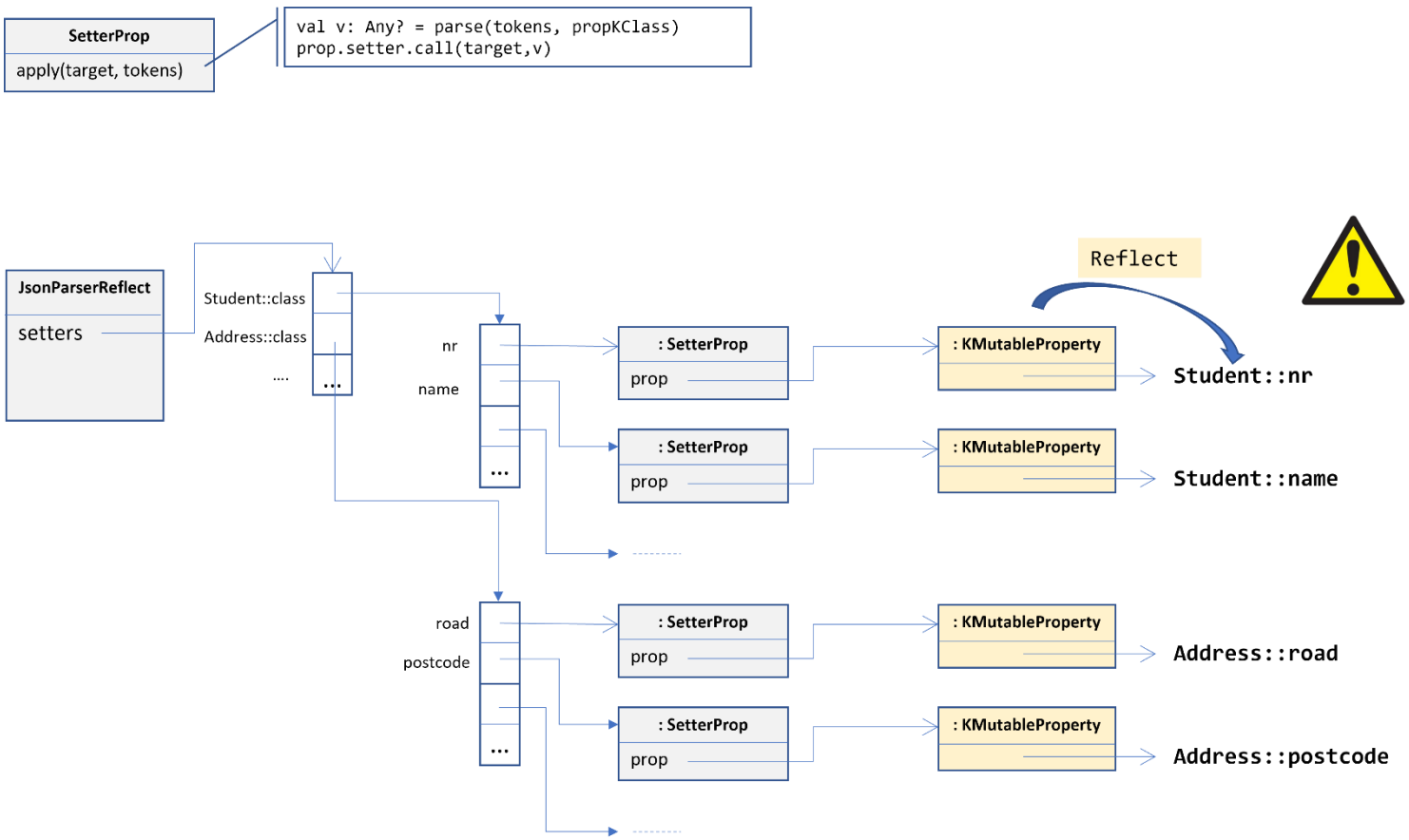
```
data class Student (var nr: Int = 0, var name: String? = null,
@JsonConvert(JsonToDate::class) val birth: Date)
```

Pode ser associado através da anotação `JsonConvert` um conversor para qualquer classe de domínio. `JsonParserReflect` tem em consideração esta anotação na inicialização das instâncias de `Setter`.

A classe `JsonParserDynamic` tem o mesmo comportamento de `JsonParserReflect`, mas **NÃO usa reflexão na atribuição de valores às propriedades**. Note, que **continua a ser usada reflexão na leitura** da *metadata*, deixando apenas de ser usada reflexão em operações como `<property>.setter.call(…)`. A atribuição de valores a propriedades é realizada directamente com base em código gerado em tempo de execução através da API de JavaPoet.

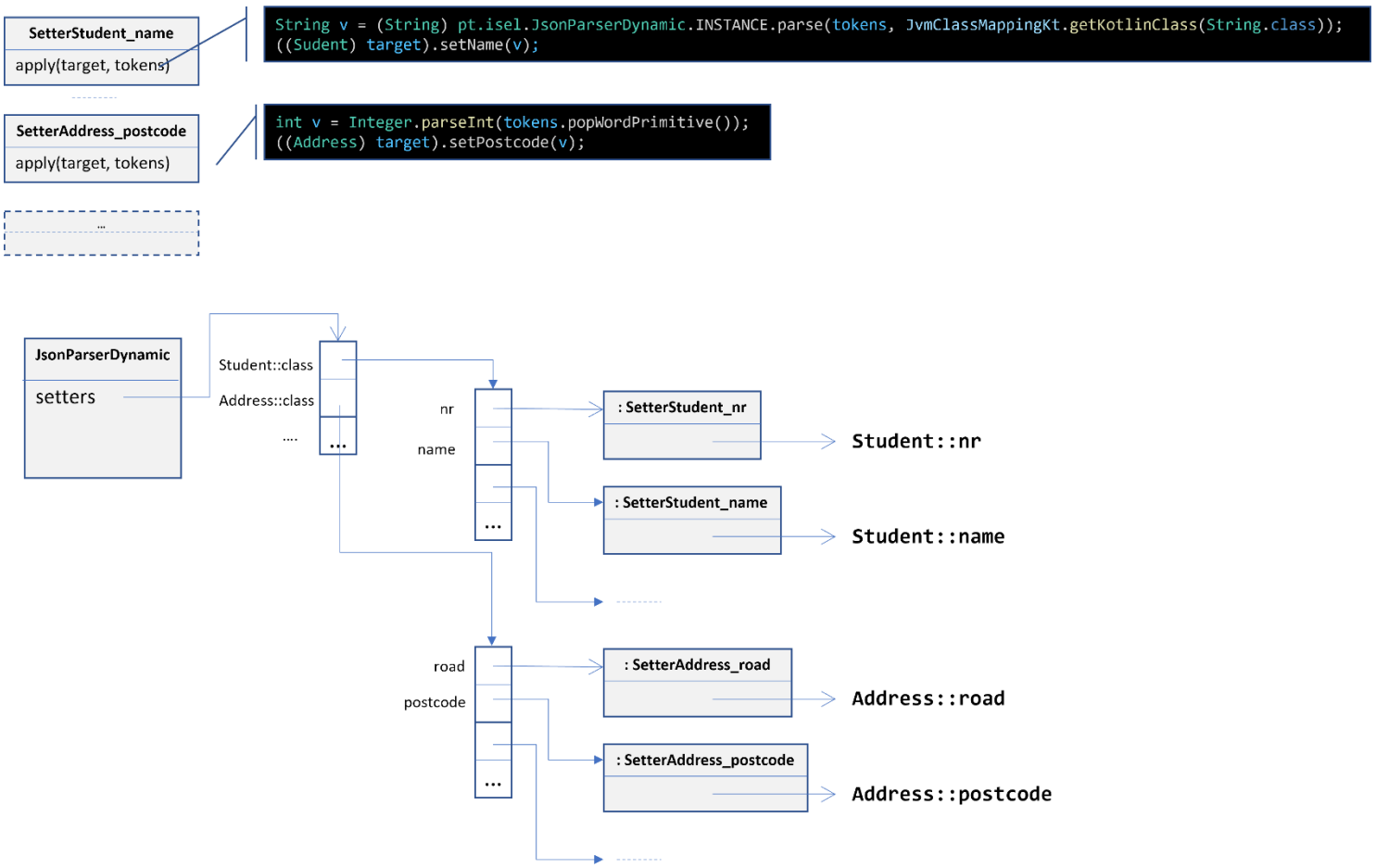
O diagrama da figura 1 apresenta um exemplo do objecto `JsonParserReflect`, onde cada instância de `SetterProp` está associada a uma instância de `KMutableProperty` que actua sobre uma determinada propriedade. A classe `SetterProp` implementa o código de afectação de uma propriedade via reflexão comum a todas as propriedades. A amarelo é evidenciado o que se pretende eliminar.

Figura 1 - Diagrama de `JsonParserReflect`



O diagrama da figura 2 apresenta um exemplo do objecto `JsonParserDynamic` onde cada propriedade tem uma implementação de `Setter` específica **em Java** (e.g. `SetterStudent_name`, `SetterAddress_postcode`), evitando o uso de reflexão na afectação de uma propriedade. Note ainda que para propriedades de tipo primitivo (e.g. `postcode`) a implementação de `Setter` evita a chamada ao `parse()` da base para não incorrer nos custos das operações de *boxing* e *unboxing*.

Figura 2 - Diagrama de `JsonParserDynamic`



`JsonParserDynamic` gera em tempo de execução implementações em Java das classes que implementam a interface `Setter` para cada propriedade.
O código das classes gerado dinamicamente através do JavaPoet é construído sobre uma instância de `com.squareup.javapoet.JavaFile`. A função `loadAndCreateInstance(source: JavaFile)` usa um `ClassLoader` para carregar a classe definida em `source` e criar uma instância dessa classe (considerando que tem um contrutor sem parâmetros).

```
import com.squareup.javapoet.JavaFile
import java.io.File
import java.net.URLClassLoader
import javax.tools.ToolProvider

private val root = File("./build")
private val classLoader = URLClassLoader.newInstance(arrayOf(root.toURI().toURL()))
private val compiler = ToolProvider.getSystemJavaCompiler()

fun loadAndCreateInstance(source: JavaFile): Any {
    // Save source in .java file.
    source.writeToFile(root)

    // Compile source file.
    compiler.run(null, null, null, "${root.path}/${source.typeSpec.name}.java")

    // Load and instantiate compiled class.
    return classLoader
        .loadClass(source.typeSpec.name)
```

```
        .getDeclaredConstructor()
        .newInstance()
}
```

A aplicação consola do projecto **jsonaif-bench** para compara o desempenho do método `parse()` entre as classes `JsonParserReflect` e `JsonParserDynamic`.

As diferenças de desempenho são observadas para diferentes tipos de objectos de domínio, tais como:

1. objectos só com propriedades de tipo primitivo (e.g. `Date`)
2. objectos só com propriedades de tipo referência.
3. objectos com propriedades de tipo primitivo e tipo referência.

As utilizações apresentadas na primeira coluna do exemplo seguinte poder ser reescritas na forma apresentada na segunda coluna:

```
JsonParser parser = ...
val student = parser.parse(json, Student::class) as Student
val p = parser.parse(json, Person::class) as Person
val ps = parser.parse(json, Person::class) as List<Person>
```

```
JsonParser parser = ...
val student = parser.parse<Student>(json)
val p : Person? = parser.parse(json)
val ps = parser.parseArray<Person>(json)
```

O método `parseSequence(json: String): Sequence<T?>` retorna uma sequência *lazy* para o *array* JSON passado por parâmetro.

Note que o novo método `parseSequence()` está disponível a ambas as implementações `JsonParserReflect` e `JsonParserDynamic`.

O método `parseFolderEager(path: String): List<T?>` e `parseFolderLazy(path: String): Sequence<T?>` retornam uma lista ou uma sequência *lazy*, onde cada elemento é o resultado de aplicar `parseObject` sobre o conteúdo de cada ficheiro presente na pasta `path`.