

# Captura de dados parametrizado por anotações em classes Java

Vítor Augusto Ueno Otto<sup>1</sup>

<sup>1</sup>Bacharelado em Ciências da Computação – Instituto Federal Catarinense Cmapus Blumenau (IFC Blumenau)

vitoruenootto@gmail.com

## 1. Descrição do Programa e Reflexão

O programa Java proposto por este trabalho coleta e analisa dados de consumo e gasto calórico. Ao iniciar o programa CLI, o usuário informa algumas informações básicas e quantas calorias é recomendado que ele consuma diariamente. Em seguida ele insere a quantidade de calorias ingeridas dia a dia referentes a uma semana inteira, que servirá de base para as operações seguintes.

Após a fase inicial de coleta de dados, o usuário pode acessar as demais funcionalidades indefinidamente, que são: cadastrar o consumo de calorias de uma semana, consultar a média de calorias ingeridas em cada semana ou realizar uma estimativa do saldo calórico após um ano, com base em uma das semanas cadastradas (Figura 1). Uma outra opção disponível ao usuário finaliza o programa ao encerrar o laço que gerencia a execução das funcionalidades.

Figura 1 – Execução da funcionalidade 2 e 3 do programa de análise de calorias

```
=====Lista de ações=====
1- Cadastrar Relatório de calorias de uma semana
2- Verificar média de calorias ingeridas para cada semana
3- Projetar saldo de calorias em um ano baseado em média
0- Sair
Digite uma opção: 2
=====

Imprimindo média de cada semana:
Média da semana 1: 2000.0

Projetando saldo de calorias em um ano:
Semanas cadastradas:
- Semana 1: [2000, 2100, 2700, 2400, 1600, 1500, 1700]
Digite o número da semana que você quer se basear: 1

Se você mantiver o padrão de consumo desta semana por um ano, você terá um saldo de -168000 calorias
isso significa que você consumiria menos calorias do que o ideal para este período
talvez seja melhor aumentar um pouco a quantidade de calorias ingeridas
```

Fonte: figura do autor (2022)

Um dos recursos que facilitaram a criação deste programa foi a reflexão, que pode ser entendido como a programação da programação. Segundo Maes (1987), um sistema reflexivo é aquele que possui estruturas que representam a si mesmo, e que quando suportadas em uma linguagem de programação, esta é tida como reflexiva. Para Tomforde et al. (2014), a reflexão é a capacidade de um sistema de monitorar a si mesmo e adaptar seu comportamento diante de incertezas, que não poderiam ser

<sup>1</sup> Repositório com os códigos do projeto:  
[https://github.com/vitorueno/POO2\\_reflexao\\_java](https://github.com/vitorueno/POO2_reflexao_java)

antecipadas no momento de design do programa. Ainda segundo Tomforde et al. (2014), na programação, a reflexão costuma se limitar a contornar restrições ou investigar estruturas dos objetos em tempo de execução.

Java é uma das linguagens que possui os recursos mais avançados em reflexão, ainda que possua limitações, como não permitir a modificação do conteúdo dos métodos nem adicionar atributos ou métodos em tempo de execução (INSA e SILVA, 2018). Segundo Tomforde et al. (2014), a reflexão do Java não ocorre por meio de uma auto-representação implementada no sistema, mas sim por meio de uma API, e se divide em introspeção (analisar os recursos disponíveis) e interseção (manipular e modificar os dados, estruturas...).

Nesse sentido, Oriel (2006) apresenta de forma resumida as classes e métodos que permitem a reflexão no Java. De acordo com ele, a maior parte dos recursos de introspecção do Java é obtido através da classe `Object` e `Field`, enquanto invocações reflexivas (“manipulação”) ocorrem por meio da classe `Method` e `Field`. Por meio destes recurso é possível se obter informações das classes (`Class.getName()`, `Class.getModifiers()`), seus métodos (`Class.getMethods()`) e atributos (`Class.getFields()`) e assim realizar atribuições (`Field.set()`), criar instâncias (`Constructor.newInstance()`) ou invocar métodos (`Method.invoke()`), por exemplo.

No programa Java deste trabalho<sup>1</sup>, a reflexão foi aplicada na primeira parte, a responsável pela coleta de dados, conforme apresentado na Figura 2. Não foi preciso solicitar campo por campo de forma engessada no código, pois por meio dos métodos reflexivos do Java foi possível ler todos os métodos da classe `Pessoa` em busca dos “sets”, que além disso precisam possuir a anotação “*@paraSolicitar*”, que além de servir como marcador, também apresentam um valor, que representa o tipo do campo, de forma a simplificar manipulação.

**Figura 2 – Código reflexivo para criação de instância de uma classe qualquer**

```
87
88 private static Object cadastrar(String nomeClasse) {
89     Object p = null;
90     try {
91         Class<?> c = Class.forName(nomeClasse); // pegando a classe
92
93         System.out.println("Cadastrando um(a) " + c.getSimpleName() + ":\n"); // informando o que estamos cadastrando
94         Scanner scanner = new Scanner(System.in);
95
96         try {
97             Constructor init = c.getDeclaredConstructors()[0];
98             try { // tentando instanciar um objeto e solicitar dados para o usuário
99                 p = (Object) init.newInstance();
100                 for (Method m : c.getDeclaredMethods()) { // lendo métodos da classe
101                     if (isSetter(m) && m.isAnnotationPresent(paraSolicitar.class)) { // deve ser solicitado ao usuário?
102                         System.out.print("Insira o seu/sua " + deSetterParaPropriedade(m.getName()) + ": ");
103                         String attrLido = scanner.next(); // pegando valor digitado pelo usuário
104                         String tipo = m.getAnnotation(paraSolicitar.class).tipo(); // pegando o valor da anotação
105
106                         if (tipo.equals("int")) { // convertendo para int caso informado pela anotação
107                             m.invoke(p, Integer.parseInt(attrLido));
108                         } else {
109                             m.invoke(p, attrLido);
110                         }
111                     }
112                 }
113             } catch (Exception e) {
114                 System.out.println("Erro ao tentar criar instância ou executar sets: ");
115                 e.printStackTrace();
116             }
117         } catch (Exception e) {
118             System.out.println("Erro ao tentar chamar o construtor: ");
119             e.printStackTrace();
120         }
121     } catch (Exception e) {
122         System.out.println(e);
123         e.printStackTrace();
124     }
125     return p;
126 }
```

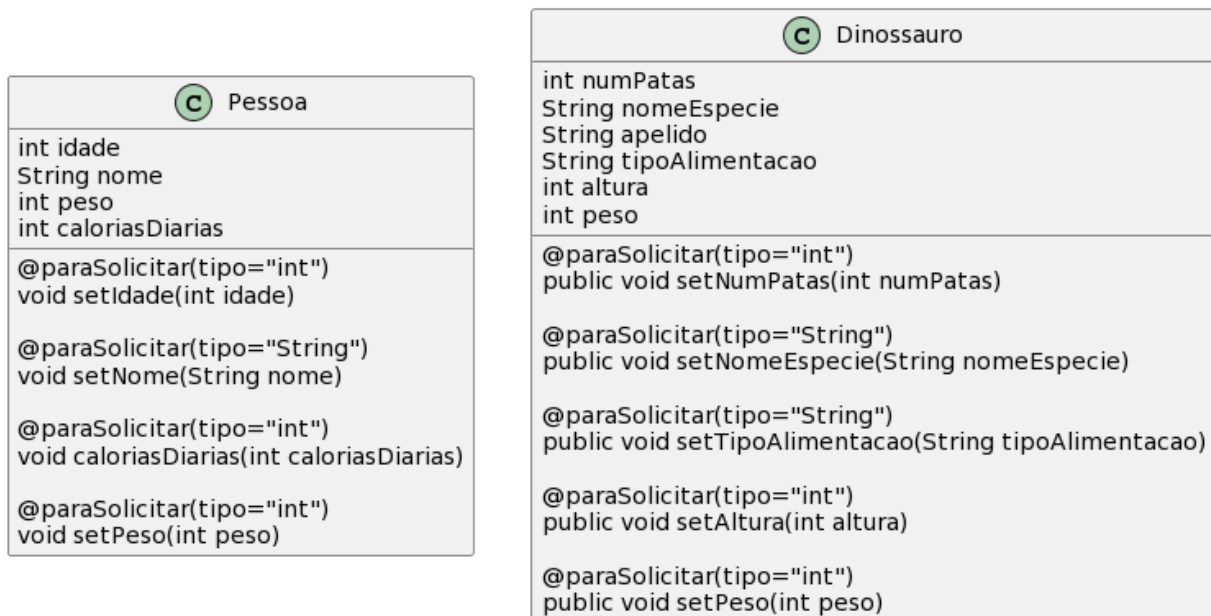
Fonte: figura do autor (2022)

O funcionamento do método cadastrar ocorre da seguinte forma: o método recebe o nome de uma classe e retorna uma instância dela com os dados informados pelo usuário. Para coletar a classe que a string representa, foi utilizado o método `forName()`. Após isso, uma mensagem exibe o nome da classe que está sendo cadastrada e em seguida um objeto é instanciado (método `new instance`). Por fim, é verificado cada método da classe (`getDeclaredMethods`) e, caso ele seja um setter (método personalizado) e possua a anotação “`paraSolicitar`” (`isAnnotationPresent`), o seu valor é solicitado ao usuário e então o respectivo set é executado com este valor (`invoke`), sendo previamente convertido para seu respectivo tipo, de acordo com o valor da anotação. Caso tudo dê certo, o objeto é então retornado, sendo preciso apenas realizar um `typecast` para o tipo final, posto que é retornado um `Object` genérico.

Como prova de conceito, uma outra classe foi criada e testada com o método “`cadastar`”, seguindo os mesmos padrões da classe `Pessoa`, conforme mostra a Figura 3. A classe `dinossauo` possui atributos do tipo `int` e `String`, dos quais alguns foram marcados com a anotação “`paraSolicitar`”. Os resultados (Figura 4) indicam que o método `cadastar` foi capaz de funcionar para ambas as classes e solicitar os dados corretos ao usuário de acordo com a classe passada por parâmetro a ela. Isso indica que o método `cadastar` é genérico o suficiente para funcionar com qualquer classe, contanto que possuam os sets e estejam marcadas com as anotações de forma correta.

**Figura 3 – Diagrama de classes – classes simplificadas**

**Classes simplificadas - Coleta de dados com reflexão**



Fonte: figura do autor (2022)

**Figura 4 – Execução da reflexão para duas classes diferentes**

```
Classes disponiveis:
1 - Pessoa
2 - Dinossauro

Digite o número de uma das classes: 2

Cadastrando um(a) Dinossauro:

Insira o seu/sua numPatas: 4
Insira o seu/sua nomeEspecie: trex
Insira o seu/sua apelido: rex
Insira o seu/sua tipoAlimentacao: carne
Insira o seu/sua altura: 2
Insira o seu/sua peso: 500

Classes disponiveis:
1 - Pessoa
2 - Dinossauro

Digite o número de uma das classes: 1

Cadastrando um(a) Pessoa:

Insira o seu/sua nome: vitor
Insira o seu/sua idade: 19
Insira o seu/sua pesoKG: 65
Insira o seu/sua quantidadeCaloriasRecomendadasPorDia: 2500
```

Fonte: figura do autor (2022)

A principal vantagem desta abordagem é a manutenção e reutilização de código. Caso a classe Pessoa precise ser trocado por outra que siga os padrões apresentados, como exemplificado pela classe Dinossauro, basta trocar o parâmetro do método cadastrar para o da nova classe e, em seguida, realizar um typecast para ter a instância do tipo Dinossauro. Desta forma, mesmo que a classe tenha cem atributos, não é preciso declarar o input para cada.

Nota-se, portanto, que a reflexão é um recurso poderoso para lidar com incertezas de tipo em tempo de execução e generalizar o código para reaproveitá-lo. Destaca-se ainda que para o contexto deste trabalho foram considerados apenas atributos do tipo inteiro e String, e a coleta de dados apenas por meio de teclado, mas os mesmos conceitos poderiam ser expandidos para abranger casos ainda mais genéricos, como o uso de interfaces gráficas e tipos personalizados de dados, abrindo uma gama ainda maior de possibilidades.

## References

- INSA, David; SILVA, Josep. Automatic assessment of Java code. **Computer Languages, Systems & Structures**, v. 53, p. 59-72, 2018. Disponível em: <https://www.sciencedirect.com/science/article/abs/pii/S1477842417301045>. Acesso em: 10 abr. 2022.
- MAES, Pattie. Concepts and experiments in computational reflection. **ACM Sigplan Notices**, v. 22, n. 12, p. 147-155, 1987. Disponível em: <https://dl.acm.org/doi/abs/10.1145/38807.38821>. Acesso em: 10 abr. 2022.
- ORIOLO, Manuel. Techniques of Java Programming: Reflection. **Techniques of Java Programming Lecture notes, Department of Computer Science, ETH Zurich, Switzerland**, 2006. Disponível em: <http://se.inf.ethz.ch/old/teaching/ss2006/0284/book/Lect5.pdf>. Acesso em: 10 abr. 2022.
- TOMFORDE, Sven et al. "Know Thyself"-Computational Self-Reflection in Intelligent

Technical Systems. In: **2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops**. IEEE, 2014. p. 150-159. Disponível em: <https://ieeexplore.ieee.org/abstract/document/7056371>. Acesso em: 10 abr. 2022.