1 Introdução

Neste material, iremos utilizar recursos de programação orientada a objetos para desenvolver um jogo. Nele, temos um personagem que se encontra isolado em uma ilha e precisa realizar atividades para se manter vivo.

2 Desenvolvimento

2.1 (Classe para descrever o personagem) Começamos descrevendo o personagem do jogo. Isso deve ser feito por meio da definição de uma **classe**. Uma classe é uma descrição ou modelo de algo de interesse para o sistema sendo implementado. Para o nosso jogo, no momento, temos apenas uma classe. Ela descreve algumas características e comportamentos do personagem. Tecnicamente, quando criamos uma classe estamos criando um novo tipo de dado, o que quer dizer que podemos declarar variáveis (e construir objetos) daquele tipo. Veja a primeira versão da classe que descreve o personagem do jogo na Listagem 2.1.1.

Listagem 2.1.1

```
public class Personagem {
  String nome;
  int energia;
  int fome;
  int sono;

  void cacar (){
    System.out.println(nome + " cacando");
  }

  void comer (){
    System.out.println(nome + " comendo");
  }

  void dormir (){
    System.out.println(nome + " dormindo");
  }
}
```

2.2 (Construindo um personagem) Uma vez feita a descrição, podemos construir um personagem propriamente dito. Para isso, vamos escrever uma classe diferente, cuja finalidade será abrigar o método main. Assim, mantemos a classe Personagem altamente coesa, já que ela permanece com uma única responsabilidade, resolvendo somente um problema (descrever o que é um personagem). Veja um teste inicial na Listagem 2.2.1.

Listagem 2.2.1

```
public class Jogo {
  public static void main(String[] args) {
    Personagem cacador = new Personagem();
    cacador.nome = "John";
    cacador.cacar();
    cacador.comer();
    cacador.dormir();
  }
}
```

2.3 (Regras para o jogo) Na classe que descreve o personagem do jogo, estamos utilizando as variáveis energia, fome e sono para representar seu nível de energia, o quão faminto ele está e seu nível de sono, respectivamente. Faz sentido, portanto, que eles sejam alterados conforme os métodos cacar, comer e dormir são utilizados. Vamos implementar as seguintes regras na classe Personagem.

- As variáveis energia, fome e sono somente admitem a atribuição de valores no intervalo [0, 10].

- O personagem começa com valores 10, 0, 0 para energia, fome e sono, respectivamente. Ou seja, ele começa com o máximo de energia, sem fome e sem sono.

- Quando o personagem caça, ele gasta dois pontos de energia. Ele somente pode caçar caso tenha pelo menos dois pontos de energia. Em qualquer caso, seus níveis de fome e sono sobem de um ponto.

- Quando o personagem come, ele reduz de 1 ponto seu nível de fome. Além disso, seu nível de energia aumenta de 1. Ele somente come se tiver nível de fome maior ou igual a 1.

- Quando o personagem dorme, ele reduz de 1 ponto seu nível de sono. Além disso, seu nível de energia aumenta de 1. Ele somente dorme se tiver nível de sono maior ou igual a 1.

Veja a Listagem 2.3.1.

Listagem 2.3.1

```
public class Personagem {
  String nome;
  int energia = 10;
  int fome = 0;
  int sono = 0;

  void cacar (){
    if (energia >= 2){
      System.out.println(nome + " cacando");
      energia -= 2;
    }
    else{
      System.out.println(nome + " sem energia para cacar");
    }
    fome = Math.min (fome + 1, 10);
    sono = Math.min (sono + 1, 10);

  }

  void comer (){
    if (fome >= 1){
      System.out.println(nome + " comendo.");
      energia = Math.min (energia + 1, 10);
      fome -= 1;
    }
    else{
      System.out.println(nome + " sem fome.");
    }
  }

  void dormir (){
    if (sono >= 1){
      System.out.println(nome + " dormindo.");
      sono -= 1;
      energia = energia + 1 <= 10 ? energia + 1 : 10;
    }
    else{
      System.out.println(nome + " sem sono.");
    }
  }
}
```

- A seguir, utilizamos a classe Jogo para testar as funcionalidades. Para ver alguma variação interessante, vamos colocar as chamadas aos métodos em um loop. Para que os resultados possam ser analisados (e não apareçam muito rapidamente na tela), podemos utilizar o método **sleep** da classe **Thread**. Para isso, vamos nos livrar da exceção verificada (assunto para ser estudado no futuro) que ele lança na assinatura do método main. Veja a Listagem 2.3.2.

Listagem 2.3.2

```
public class Jogo {
  public static void main(String[] args) throws InterruptedException {
    Personagem cacador = new Personagem();
    cacador.nome = "John";

    while (true){
      cacador.cacar();
      cacador.comer();
      cacador.dormir();
      cacador.cacar();
      cacador.cacar();
      cacador.cacar();
      System.out.println("====================");
      Thread.sleep(2000);
    }
  }
}
```

- Podemos, inclusive, adicionar um outro personagem ao jogo. Digamos que ele costume dormir um pouco mais do que o primeiro. Veja a Listagem 2.3.3.

Listagem 2.3.3

```
public class Jogo {
  public static void main(String[] args) throws InterruptedException {
    Personagem cacador = new Personagem();
    Personagem soneca = new Personagem ();
    soneca.nome = "Soneca";
    cacador.nome = "John";

    while (true){
      cacador.cacar();
      soneca.dormir();
      cacador.comer();
      soneca.dormir();
      cacador.dormir();
      soneca.dormir();
      cacador.cacar();
      soneca.comer();
      cacador.cacar();
      soneca.cacar();
      System.out.println("====================");
      Thread.sleep(3000);

    }
  }
}
```

2.4 (Encapsulamento) O funcionamento do jogo depende do estado em que cada personagem se encontra. O estado de um objeto é caracterizado pelos valores que ele possui em suas variáveis de instância. No momento, qualquer classe tem acesso direto às variáveis energia, fome e sono e, portanto, têm a possibilidade de comprometer o estado de objetos existentes no sistema. É uma excelente prática fazer com que **toda classe seja responsável pela manutenção do estado de seus objetos**. Desejamos que, a qualquer momento, os objetos do sistema estejam em estado consistente e desejamos que essa verificação seja feita pelas classes a partir das quais foram construídos. Quando uma classe **encapsula** seus detalhes de implementação, ela restringe o acesso a eles. As demais classes não mais podem acessá-los diretamente. Assim, a consistência do estado de cada objeto fica sob responsabilidade de sua própria classe. Em Java, para encapsular detalhes de implementação de uma classe, utilizamos o **modificador de acesso** private. Veja a Listagem 2.4.1.

Listagem 2.4.1

```
private int energia = 10;
private int fome = 0;
private int sono = 0;
```

- Tente acessar os atributos a partir da classe Jogo e veja o erro que o compilador reporta.

2.5 (Construtores) Digamos que desejamos permitir que os valores iniciais para as variáveis energia fome e sono sejam especificados pelas classes que utilizam a classe Personagem. No entanto, isso somente será permitido caso os valores estejam dentro do intervalo especificado. Caso contrário, a classe Personagem mantém os valores padrão. Para isso, podemos manter as variáveis encapsuladas e ofertar um construtor público que recebe os valores por meio de sua lista de parâmetros. Veja a Listagem 2.5.1.

Listagem 2.5.1

```
public Personagem (int energia, int fome, int sono){
   if (energia >= 0 && energia <= 10)
     this.energia = energia;
   if (fome >= 0 && fome <= 10)
     this.fome = fome;
     if (sono >= 0 && sono <= 10)
     this.sono = sono;

 }
```

- Na classe Jogo, o uso do construtor pode ser feito como a Listagem 2.5.2 ilustra.

Listagem 2.5.2

```
Personagem cacador = new Personagem(10, 0, 0);
Personagem soneca = new Personagem (2, 6, 4);
```

- Podemos, também, especificar um construtor que recebe o nome do personagem, além dos demais valores. Assim, podemos encapsular a variável nome também. Veja a Listagem 2.5.3.

Listagem 2.5.3

```
private String nome;
public Personagem(String nome, int energia, int fome, int sono) {
    this.nome = nome;
    if (energia >= 0 && energia <= 10)
      this.energia = energia;
    if (fome >= 0 && fome <= 10)
      this.fome = fome;
    if (sono >= 0 && sono <= 10)
      this.sono = sono;

  }
```

- Ajuste o código cliente (classe Jogo) para que ela não mais acesse a variável nome diretamente.

- Perceba, contudo, que há muito código duplicado. Podemos fazer com que um construtor chame outro e, assim, escrever as validações uma única vez. Veja a Listagem 2.5.4.

Listagem 2.5.4

```
public Personagem(String nome, int energia, int fome, int sono) {
    this (energia, fome, sono);
    this.nome = nome;
}
```

Exercícios

1. Adicione um método à classe Personagem que exibe o estado (valores de energia, fome e sono) dos objetos. Chame ele em cada método existente na sua classe. Ele deve exibir o nome do personagem seguido dos valores de cada variável.

2. Adicione um novo personagem e inclua ele no “loop do jogo”.

Referências

DEITEL, P. e DEITEL, H. **Java Como Programar**. 8ª Edição. São Paulo, SP: Pearson, 2010.

LOPES, A. e GARCIA, G. **Introdução à Programação – 500 Algoritmos Resolvidos**. 1ª Edição. São Paulo, SP: Elsevier, 2002.