Capítulo 3

Encapsulamento e Métodos Construtores

Se você cria uma criança para acreditar que tudo são raios de sol e flores, ela entrará no mundo real e morrerá. Essa é a razão dos contos de fadas serem tão sinistros, porque precisamos encapsular essas coisas para nos protegermos delas, para assim quando confrontamos o horror genuíno que é o dia-a-dia, não enlouquecermos.
Joss Whedon, roteirista, diretor e produtor de cinema e TV.

Este Capítulo tem por objetivo apresentar os conceitos de métodos construtores e de encapsulamento.

3.1 Métodos Construtores

As linguagens que disponibilizam recursos para a programação Orientada a Objetos geralmente utilizam-se de métodos específicos para "construir" um objeto. Estes métodos são acionados de forma automática pela própria linguagem quando um objeto é instanciado (criado). Desta forma, os desenvolvedores podem fazer uso deles para manipular ações que possam ser necessárias durante esses eventos. Este método especial é chamado de Construtor (relacionado ao momento que um objeto é instanciado).

O Exemplo 3.1 descreve uma classe $Pessoa$ que contém 2 atributos: $nome$ e $idade$; e dois métodos: $fazAniversario()$ e $imprime()$. A linha 16 instancia um objeto da classe Pessoa e armazena na variável p. As atribuições dos valores iniciais para os atributos são

realizadas pelas linhas 17 e 18. Note que toda vez que um novo objeto for instanciado será necessário repetir o processo de atribuição de alguns valores iniciais.

```
01  class Pessoa {
02    int idade;
03    String nome;
04    void fazAniversario() {
05      idade++;
06    }
07    void imprime() {
08      System.out.println("Nome:"+nome);
09      System.out.println("Idade:"+idade);
10    }
11  }
12
13  class Prog {
14    public static void main(String args[]) {
15      Pessoa p;
16      p = new Pessoa();
17      p.nome = "Ana";
18      p.idade = 15;
19      p.fazAniversario();
20      p.imprime();
21    }
22  }
```

Exemplo 3.1: Implementação da classe Pessoa.

A inicialização de um objeto pode ser tratada utilizando o conceito de métodos construtores. No Exemplo 3.2, a classe Pessoa foi modificada adicionando um método com o mesmo nome da classe. No Java, todo método definido com o mesmo nome da classe passa a ser caracterizado como método construtor. Assim, quando o objeto for instanciado, por exemplo pela linha 20, este método é executado tendo como finalidade iniciar os valores dos atributos.

O método Construtor pode ou não ser definido em uma Classe. Se for definido, ele será sempre executado quando um objeto estiver sendo instanciado. Caso não exista, o objeto é instanciado normalmente sem nenhum tratamento.

```
01  class Pessoa {
02    int idade;
03    String nome;
04    Pessoa() {
05      nome = "sem nome";
06      idade = 0;
07    }
08    void fazAniversario() {
09      idade++;
10    }
11    void imprime() {
12      System.out.println("Nome:"+nome);
13      System.out.println("Idade:"+idade);
14    }
15  }
16
17  class Prog {
18    public static void main(String args[]) {
19      Pessoa p;
20      p = new Pessoa();
21      p.fazAniversario();
22      p.imprime();
23    }
24  }
```

Exemplo 3.2 : Implementação da classe Pessoa.

Embora seja um método como os outros, ele não pretende devolver nenhuma informação, e desta forma, deve ser encarado como procedimento[1] e não uma função [2].

Apesar da declaração do método Construtor não ser obrigatória, a utilização dele nos garante maior controle e clareza na reutilização do código.

O método construtor, quando definido pela classe, disponibiliza uma forma de controlar as características de criação do objeto. Isto é interessante em casos onde queremos definir valores para atributos, instanciar outras classes, executar outros métodos e até mesmo permitir uma alocação dinâmica de memória que poderia ser utilizada durante a vida do objeto.

[1] Procedimento é um bloco de código que desempenha alguma funcionalidade e, por padrão, não retorna valor após seu processamento.

[2] Função difere de procedimento por devolver valor após seu processamento.

A sintaxe utilizada na declaração do método construtor segue a mesma usada na definição de métodos comuns, com a diferença que o tipo de retorno não é especificado.

Um exemplo de uso do método Construtor se dá quando queremos nos aproveitar do fato dele sempre ser executado no momento em que uma classe é instanciada, usando este recurso para atribuir valores iniciais aos atributos do objeto que está sendo criado. Assim, os objetos instanciados desta classe podem assumir um comportamento inicial padrão para todos os objetos da mesma classe, ou seja, os objetos serão criados com todos ou alguns atributos já valorados.

```
01  class Pessoa {
02    int idade;
03    String nome;
04    Pessoa(String nome, String idade) {
05      this.nome = nome;
06      this.idade = idade;
07    }
08    void fazAniversario() {
09      idade++;
10    }
11    void imprime() {
12      System.out.println("Nome:"+nome);
13      System.out.println("Idade:"+idade);
14    }
15  }
16
17  class Prog {
18    public static void main(String args[]) {
19      Pessoa p;
20      p = new Pessoa("Ana",15);
21      p.fazAniversario();
22      p.imprime();
23    }
24  }
```

Exemplo 3.3: Implementação da classe Pessoa.

Para criar um objeto no qual os valores iniciais de seus atributos sejam passados no momento de sua criação, ou seja, para instanciar a classe passando os valores iniciais para seus atributos, devemos declarar parâmetros a serem recebidos pelo método Cons-

trutor. Assim, os valores iniciais são passados no momento em que instanciamos uma classe e são recebidos pelos parâmetros do método Construtor podendo ser atribuídos aos respectivos atributos, conforme apresentado pelo Exemplo 3.3.

Além de poder criar objetos com valores iniciais para seus atributos, é uma boa prática, utilizar o método construtor, para controlar a criação dos objetos e garantir sua funcionalidade e acessibilidade. Em alguns casos, o método construtor pode ser utilizado para instanciar outras classes, agregando objetos conforme será abordado em capítulos futuros. Podemos também utilizar o método Construtor para permitir uma alocação dinâmica de memória que poderia ser utilizada durante a vida do objeto. Na verdade, este é o recurso que a maioria das linguagens utiliza para permitir o acesso ao momento em que um objeto esta sendo instanciado de uma classe.

3.2 Encapsulamento

Na linha 20 do Exemplo 3.3, após o objeto ser instanciado da classe Pessoa e ser armazenado na variável p, note que qualquer valor poderia ser atribuído ao atributo $idade$ do objeto, como por exemplo `p.idade = -5;`. Apesar de percebermos que no exemplo citado nenhum problema de execução possa ocorrer, dependendo da situação e do comportamento planejado pela classe para um futuro objeto da classe Pessoa, a consistência da informação armazenada no atributo $idade$, para o caso de valores negativos, pode trazer conflito na funcionalidade do objeto. Desta forma, seria interessante neste caso, evitar que valores inválidos sejam atribuidos.

O conceito de Encapsulamento é o processo no qual nenhum acesso direto aos dados é concedido, sendo possível a sua manipulação somente por meio da interação com o objeto. O Encapsulamento é o mecanismo que provê proteção de acesso aos membros internos de um objeto.

Encapsulamento vem de encapsular, que em programação orientada a objetos significa separar o programa em partes, mais isoladas possível. A idéia é tornar o software mais flexível e fácil de modificar e de criar novas implementações. Em resumo, encapsulamento se refere a uma forma de restringir acesso aos recursos do objeto. O conceito de encapsulamento garante que os objetos disponibilizem métodos que permitem que outros objetos se relacionem com ele, sem que seus atributos possam ser manipulados de forma direta.

No paradigma da Orientação a Objetos, a reutilização de código é uma característica que o desenvolvedor deve considerar ao projetar suas classes. O encapsulamento facilita este processo e deve ser utilizado buscando garantir que o objeto possa ser reaproveitado em outros projetos.

Os modificadores de acesso implementam o conceito de encapsulamento por meio de palavras-chaves. Os modificadores que o paradigma da Orientação a Objetos conceitua são:

- PUBLIC - Publico
- PRIVATE - Privado
- PROTECTED - Protegido

3.2.1 Modificador Public

Por padrão no Java, todo atributo ou método que não esteja declarado como protected ou private, é considerado *public*.

Membros declarados como público (*public*) poderão ser acessados livremente a partir da própria classe, por classes especialistas e por outros trechos fora do escopa da classe que dela se utilizam. A utilização do *public* não restringe acesso aos atributos e métodos

declarados como tal. Desta forma, os objetos que se relacionam com um objeto A, que têm seus atributos / métodos declarados como $public$, podem arbitrariamente manipular os atributos do objeto A e chamar seus métodos. Está prática deve ser evitada, pois ao programar orientado a objetos, deve-se seguir o conceito do paradigma da Orientação a Objeto, onde o reuso do código é um ponto importante, e a utilização do encapsulamento facilita o reaproveitamento do código.

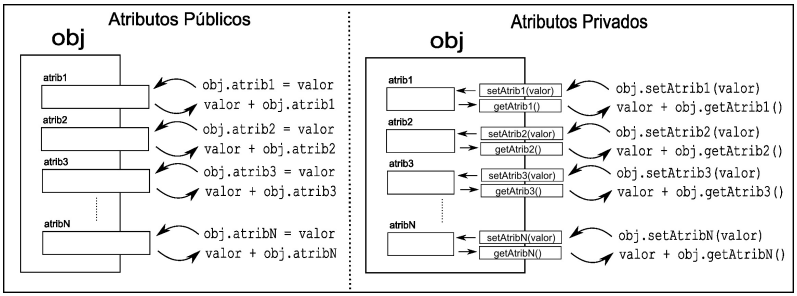

Figura 3.1: Representação do conceito de encapsulamento.

3.2.2 Modificador Private

Está é a visibilidade mais restrita do conceito de encapsulamento, ou seja, somente o próprio objeto pode acessar seus atributos e métodos declarados como privados $private$. Membros declarados como $private$, só podem ser acessados a partir da própria classe em que foram declarados, nem seus descendentes e nem programas que fazem uso desta classe poderão acessar membros declarados como $private$.

Em resumo, a utilização do modificador $private$, se dá aos atributos e métodos que são essenciais ao funcionamento do objeto e que somente o próprio objeto pode manipular. Deve-se, portanto, utilizar o modificador $private$ em membros que podem alterar o funcionamento do objeto, tornado-o instável e inseguro.

Em sistemas orientados a objetos é comum desenvolvedores declararem todos os atributos como $private$ e disponibilizar métodos para manipulação destes atributos (getters e setters). O conceito do encapsulamento não obriga a utilização de todos os manipuladores,

porém cada objeto deve ser modelado buscando sua reutilização, isto implica na correta utilização dos modificadores, pois os objetos podem evoluir e sem o encapsulamento esta evolução pode impedir o reuso.

3.2.3 Modificador Protected

O modificador *protected* será descrito no Capítulo que tratará do conceito de herança.

3.3 Métodos de Acesso

A Programação Orientada a Objetos adota como padrão proteger o acesso direto aos seus atributos, declarando-os sempre com o modificador de acesso *private*. A idéia é que cada objeto seja responsável por controlar seus atributos. Desta forma, passa a ser resposabilidade dele julgar se aquele novo valor é válido ou não para o atributo em questão. Esta validação não deve ser controlada por quem está usando o objeto e sim por ele mesmo, possibilitando que o sistema esteja preparado para futuras mudanças.

O Exemplo 3.4 apresenta a classe Pessoa com os atributos nome e idade encapsulados. Além disso, foi disponibilizado os métodos de acesso e modificador para ambos os atributos: *getIdade*, *setIdade*, *getNome* e *setNome*. Adotamos como nomenclatura para estes métodos sempre começar com os identificadores *get* para acesso e *set* para a alteração dos valores, seguido dos respectivos nomes dos atributos. Estes métodos também são conhecidos como *getters* e *setters*.

Apesar de ser uma boa prática sempre definir os métodos de acesso, em alguns casos o acesso para alguns atributos podem não ser disponibilizados. Assim, não é obrigatório a definição dos métodos de acesso para todos os atributos, sendo implementado quando necessário.

```
01  class Pessoa {
02    private int idade;
03    private String nome;
04    public Pessoa(String nome, String idade) {
05      this.nome = nome;
06      this.idade = idade;
07    }
08    public void fazAniversario() {
09      idade++;
10    }
11    public void imprime() {
12      System.out.println("Nome:"+nome);
13      System.out.println("Idade:"+idade);
14    }
15    public void setIdade(int idade) {
16      this.idade = idade;
17    }
18    public int getIdade() {
19      return idade;
20    }
21    public void setNome(String nome) {
22      this.nome = nome;
23    }
24    public String getNome() {
25      return nome;
26    }
27  }
28
29  class Prog {
30    public static void main(String args[]) {
31      Pessoa p;
32      p = new Pessoa("Ana",15);
33      p.fazAniversario();
34      p.imprime();
35      p.setIdade(25);
36      if(p.getIdade() > 20)
37        p.imprime();
38    }
39  }
```

Exemplo 3.4: Atributos Nome e Idade com Métodos de acesso.

3.4 Modelando classes

Esta Seção tem como objetivo exercitar a criação de classes que utilizam métodos construtores e o conceito de encapsulamento.

Problema 3.1: *Construa uma classe CartaoTelefonico contendo como atributo a quantidade de créditos disponíveis pelo cartão. Implemente nesta classe um método construtor que receba como parâmetro a quantidade de créditos inicial do cartão, e os métodos de acesso/modificadores para o atributo da quantidade de créditos. Crie um método status que imprima a quantidade de créditos do cartão.*

Problema 3.2: *Construa uma classe Orelhao contendo como atributo um número inteiro representando a área DDD (exemplo: 11, 21,...) em que está localizado o orelhão e a quantidade de ligações realizadas pelo aparelho. Implemente nesta classe um método construtor que receba como parâmetro a área em que o orelhão está localizado. Todo objeto instanciado da classe Orelhao deve estar com a quantidade de ligações zeradas. Crie um método fazLigacao(int area, CartaoTelefonico cartao, int tempo) que receba como parâmetro a área do número do telefone a ser ligado, um objeto da classe CartaoTelefonico ,e o tempo em segundos da ligação. Este método deve ser responsável de incrementar a quantidade de ligações e atualizar o crédito do cartão. Sabe-se que uma ligação local consome 1 crédito a cada 20 segundos, e uma ligação fora da localidade, consome 1 crédito a cada 10 segundos. Ligações só devem ser realizadas para cartões com créditos suficientes para satisfazer o tempo indicado. Crie um método status que imprima a área DDD e o número de ligações realizadas.*

Problema 3.3: *Construa uma classe Piloto contendo como atributos a quantidade de horas de vôo e o nome do piloto. Implemente nesta classe um método construtor que receba como parâmetro uma string com o nome do piloto, e os métodos de acesso/modificadores apenas para o atributo do nome do piloto. Todo objeto piloto instanciado deve inicialmente estar com a quantidade de horas de vôo zerada. A classe deve disponibilizar um método adicionaHoras(int horas) que adiciona uma quantidade de horas de vôo ao piloto. Crie um método imprime*

que imprima o nome do piloto e a quantidade de horas de vôo, sendo que deverá ser impresso o tipo do piloto: até 200 horas, co-piloto, caso contrário comandante.

Problema 3.4: *Construa uma classe Aviao contendo como atributo um número inteiro representando a quantidade de horas de atividade do avião e implemente nesta classe um método construtor que inicialize este atributo com zero. Crie um método fazVoo(int horas, Piloto piloto) que receba como parâmetro a quantidade de horas do vôo, e um objeto da classe Piloto. Este método deve ser responsável de incrementar a quantidade de horas de atividade do avião e atualizar a quantidade de horas de vôo do piloto. Sabe-se que um avião faz uma revisão a cada 200 horas de atividade. Vôos só devem ser realizados para aviões que tenham feito revisões a cada 200 horas. Crie um método fazRevisao() que zera a quantidade de horas de atividade do avião e um método status() que imprima a quantidade de horas de atividade e se o avião precisa ou não de revisão.*

Problema 3.5: *Construa uma classe BilheteEvento contendo um atributo inteiro para controlar se o bilhete já foi usado para entrada (valor 1) e saída (valor 2) em um evento . Implemente nesta classe um método construtor que inicialize este atributo com zero, e um método de acesso que retorne o valor deste atributo. A classe deve disponibilizar um método entrada(), e um método saida(), para registrar a entrada e saída. Crie um método imprime que imprima qual a situação do bilhete.*

Problema 3.6: *Construa uma classe CatracaEvento contendo como atributo um número inteiro representando a quantidade de pessoas presentes no evento e implemente nesta classe um método construtor que inicialize este atributo com zero. Crie um método registraEntrada(BilheteEvento bilhete) e um método registraSaida(BilheteEvento bilhete) ambos recebendo como parâmetro um objeto da classe BilheteEvento. Estes métodos devem ser responsáveis de registrar a entrada e saída dos bilhetes e atualizar a quantidade de pessoas presentes no evento. Sabe-se que um bilhete com entrada já registrada não pode realizar novas entradas, como também bilhetes com registro de saída não podem sair e nem entrar novamente. A quantidade de pessoas presentes*

no evento é contabilizada pelo número de bilhetes que deram entrada e que não registraram saída. Crie um método imprime que imprima a quantidade de pessoas no evento.

Problema 3.7 : *Construa uma classe TimeFutebol contendo como atributos: o nome do time, um inteiro que representa a quantidade de jogos realizados pelo time, um inteiro que indica o número de pontos ganhos, e um inteiro que segue a seguinte representação: 0 – retranqueiro, 1 – joga no ataque. Implemente nesta classe um método construtor que receba como parâmetro uma string com o nome do time e um inteiro indicando se o time é retranqueiro ou joga no ataque. Este método construtor deve inicializar a quantidade de jogos e os pontos ganhos com zero. Crie apenas um método de acesso para retornar o valor do estilo de jogo do time. A classe deve disponibilizar um método venceu(), um método empatou(), e um método perdeu() para registrar respectivamente a vitória (3 pontos), empate (1 ponto), e derrota (0 pontos), atualizando devidamente os pontos ganhos e jogos realizados. Crie um método imprime que imprima qual a situação do time.*

Problema 3.8 : *Construa uma classe Campeonato contendo como atributo um número inteiro representando a quantidade de jogos realizados e implemente nesta classe um método construtor que inicialize este atributo com zero. Crie um método realizaJogo(TimeFutebol timeCasa, TimeFutebol timeVisitante) que recebe como parâmetro dois objeto da classe TimeFutebol. Este método é responsável em analisar e registrar o resultado do jogo para cada time. Sabe-se que um time retranqueiro sempre ganha quando joga na casa de um adversário que joga no ataque. Times que jogam no ataque em jogos na casa do adversário sempre ganham. As outras combinações, sempre resultam em empate. Crie um método imprime() que imprima a quantidade de jogos realizados no campeonato.*

Problema 3.9 : *Construa uma classe Roupa contendo um atributo para armazenar o peso em Kg da roupa e um atributo para controlar se ela esta suja ou limpa. Implemente nesta classe um método construtor que receba como parâmetro o peso da roupa. Todo objeto instanciado da classe Roupa deve estar limpo. Crie os métodos de acesso/modificadores*

para o atributo de peso e disponibilize dois métodos: suja() e limpa() para alterar o estado do atributo que controla (roupa suja / limpa). Crie um método status que imprima o peso da roupa e seu estado de limpeza.

Problema 3.10 : *Construa uma classe MaquinaDeLavar contendo um atributo para armazenar a capacidade máxima em Kg e um atributo para armazenar a quantidade de horas de uso da máquina. Implemente nesta classe um método construtor que receba como parâmetro a capacidade máxima da máquina. Todo objeto instanciado da classe MaquinaDeLavar deve estar com a quantidade de horas de uso zerada. Crie um método limpaRoupa(Roupa roupa) que receba como parâmetro um objeto da classe Roupa,e que realize a tarefa de lavar a roupa, ou seja, este método deve ser responsável de incrementar a quantidade de horas de uso da máquina e atualizar o objeto roupa como limpa. Sabe-se que para cada kg de roupa a máquina demora 30 minutos, além de somente lavar roupas com peso menor que sua capacidade máxima. Crie um método status que imprima sua capacidade máxima e a quantidade de horas de uso.*

Problema 3.11 : *Construa uma classe Passageiro contendo um atributo para armazenar o nome do passageiro e um atributo para armazenar a quantidade de dinheiro do passageiro. Implemente nesta classe um método construtor que receba como parâmetro o nome e a quantidade inicial de dinheiro do passageiro. Crie os métodos de acesso/modificadores para o nome do passageiro e disponibilize um método pagou(float valor) que representa a ação do passageiro pagar um certo valor a alguém, ou seja, este método subtraí o valor da quantidade de dinheiro do mesmo. Crie um método status que imprima o nome do passageiro e a quantidade de dinheiro do passageiro.*

Problema 3.12 : *Construa uma classe Taxista contendo um atributo para armazenar o nome do Taxista e um atributo para armazenar seu lucro. Implemente nesta classe um método construtor que receba como parâmetro o nome do taxista. Todo objeto instanciado da classe Taxista deve estar com seu lucro zerado. Crie um método fazCorrida(Passageiro passageiro, int bandeira) que receba como parâmetro um objeto da classe Passageiro e um inteiro representado (1 – Bandeira Um, 2 – Bandeira Dois), que realiza a viagem do passageiro, ou seja, este método deve ser*

responsável de incrementar o lucro do taxista e atualizar a quantidade de dinheiro do objeto passageiro com o valor da corrida. Sabe-se que taxistas cobram R$20,00 para viagens na Bandeira Um e R$50,00 na Bandeira 2. Crie um método status que imprima o nome e o lucro do taxista.

Problema 3.13 : *Construa uma classe Motorista contendo um atributo para armazenar o nome do motorista, um atributo char para armazenar o estado de sua carteira de habilitação ('R' reguralizada, 'V' vencida e 'S' suspensa), um atributo para armazenar o valor em R$ de possíveis multas recebidas, e um atributo para controlar se o motorista é do tipo que oferece ou não propinas para guardas. Todo objeto instanciado da classe Motorista deve estar com o valor de multas zerado. Implemente nesta classe um método construtor que receba como parâmetro o nome do motorista, um valor char com o estado da carteira de habilitação e um valor true/false representando o motorista que paga ou não propinas. Crie apenas os métodos de acesso (get) para os atributos (estado da carteira de habilitação e do pagamento de propina). Além disso, disponibilize dois métodos: venceHabiltacao() e suspendeHabiltacao() para alterar o estado do atributo que controla o estado da carteira de habilitação e um método recebeMulta(float valor) que acumula os valores das multas recebidas. Crie um método imprime() que imprima o o nome do motorista, o estado da carteira de habilitação, o valor de multas e se ele paga ou não propinas.*

Problema 3.14 : *Construa uma classe Guarda contendo um atributo para armazenar o nome do guarda, um atributo para controlar se o guarda é corrupto ou não, e um atributo para armazenar a quantidade de propinas aceitas pelo guarda. Implemente nesta classe um método construtor que receba como parâmetro o nome do guarda e um valor true/false representando se o guarda é corrupto. Todo objeto instanciado da classe Guarda deve estar com a quantidade de propinas zerada. Crie apenas o método de acesso (get) para o atributo que indica se o guarda é corrupto. Além disso, crie um método fiscaliza(Motorista motorista) que receba como parâmetro um objeto da classe Motorista, e que realize a tarefa de fiscalizar o motorista, ou seja, este método deve ser responsável de verificar se a carteira de habilitação do motorista está regularizada. Caso a habilitação esteja vencida, o motorista recebe uma multa de R$ 70,00. Caso a habilitação esteja suspensa, a multa é de R$ 180,00. Quando o motorista é do tipo que oferece propinas, dependendo*

do tipo do guarda temos: para guardas corruptos, o motorista não recebe multas, caso contrário, o motorista tem sua carteira suspensa e paga multa de R$300,00. No caso do guarda corrupto, incrementa-se a quantidade de propinas recebidas pelo guarda. Crie um método imprime() que imprima o nome do guarda, a quantidade de propinas recebidas e se ele é corrupto ou não.

Problema 3.15 : *Construa uma classe Blitz contendo um atributo para armazenar a quantidade de fiscalizações realizadas, um atributo para armazenar a quantidade de motoristas com carteiras regularizadas e um atributo para armazenar a quantidade de fiscalizações realizadas por guardas corruptos. Implemente nesta classe um método construtor que não receba parâmetros, mas inicialize os atributos com zero. Crie um método fazFiscalizacao(Guarda guarda, Motorista motorista) que receba como parâmetro um objeto da classe Guarda e um da classe Motorista, e realize a fiscalização do motorista pelo guarda. Além disso, este método deve incrementar os valores dos atributos da classe: quantidade de fiscalizações realizadas, quantidade de motoristas com carteiras regularizadas, quantidade de fiscalizações realizadas por guardas corruptos. Crie um método imprime() que imprima as informações dos atributos.*

Problema 3.16 : *Construa uma classe Arma contendo um atributo char para armazenar o tipo da arma ('L' lança, 'E' espada e 'A' arco e flecha) e um atributo para armazenar a força de destruição da arma (5 para espada, 4 para a lança e 3 para arco e flecha). Implemente nesta classe um método construtor que receba como parâmetro um valor char com o tipo da arma. Todo objeto instanciado da classe Arma deve ter a força de destruição configurada dependendo do tipo da arma. Crie um método de acesso (get) para o atributo do tipo da arma e um para o atributo força de destruição. Crie um método imprime que imprima o tipo da arma e sua força de destruição.*

Problema 3.17 : *Construa uma classe Gladiador contendo um atributo para armazenar o nome do gladiador, um atributo que armazene a quantidade de vidas (0 até 5) e um atributo para controlar se o gladiador está usando ou não armadura. Implemente nesta classe um método construtor que receba como parâmetro o nome do gladiador e se ele deve ter armadura. Todo objeto instanciado da classe Gladiador deve ter quantidade de vidas igual a 5. Crie apenas um método de acesso (get) para o atributo de quantidade de vidas. Crie um método rece-*

beGolpe(Arma arma) que receba como parâmetro um objeto da classe Arma, e que realize a tarefa de retirar o numero de vidas dependendo da força de destruição da arma (Cada força de destruição retira uma vida, sendo que gladiadores com armadura conseguem proteger-se de 2 forças de destruição da arma, ou seja, se a arma tem 5 forças, apenas 3 vidas são retiradas, para 4 forças retiram-se 2 vidas e para 3 forças, 1 vida. Um gladiador estará morto caso seu número de vidas seja zero. Crie um método imprime que imprima o nome do gladiador, quantidade de vidas, se está de armadura e se continua vivo ou morto.

Problema 3.18 : *Construa uma classe Arena contendo um atributo para armazenar a quantidade de torcedores da arena e um atributo para controlar se os torcedores estão felizes ou tristes. Implemente nesta classe um método construtor que receba como parâmetro a quantidade de torcedores da arena. Todo objeto instanciado da classe Arena deve estar com seus torcedores contentes. Crie um método realizaCombate(Gladiador glad1, Arma arma1, Gladiador glad2, Arma arma2) que receba como parâmetro dois objetos da classe Gladiador e dois objetos da classe Arma, e que realize a tarefa de realizar o combate entre os gladiadores (glad1, glad2) e suas respectivas armas (arma1, arma2), ou seja, este método deve ser responsável de efetuar os golpes em cada gladiador com a arma do adversário. Quando os dois gladiadores continuam vivos após um combate, os torcedores ficam tristes e 25% deles deixam a arena. Caso contrário, eles ficam felizes e a arena sofre um acréscimo de 10% de torcedores. Crie um método imprime que imprima a quantidade de torcedores da arena e se eles estão contentes ou tristes.*