OO - Lorenzon

Encapsulamento

Capacidade de proteger suas propriedades e métodos de forma que nenhum agente externo tenha acesso a elas sem solicitá-las.

O encapsulamento provê visibilidade ou ocultação de atributos e métodos.

Duas regras regem o encapsulamento:

- **Regra 1: Apenas os métodos devem ser capazes de modificar o estado interno de um objeto;**
- **Regra 2: A resposta a uma mensagem deve ser completamente determinada pelo estado interno do objeto receptor.**

Visibilidade

A visibilidade se dá por modificadores que são aplicados às classes, atributos e métodos. Temos 3 tipos de especificadores de visibilidade e suas aplicações nas categorias definidas:

Especificador	Definição	Classe	Atributo	Método
+ public	Público	Torna a classe pública.	Torna o atributo público.	Torna o método público
- private	Ocultação interna	A classe é privada ao escopo de sua definição.	O atributo é provado da classe.	O método é privado da classe.
# protected	Visibilidade hereditária.	A classe é visível no pacote ou na herança.	O atributo é visível no pacote e na herança.	O atributo é visível no pacote e na herança.
~~default~~	Sem especificador	Mesmo que protected.	Mesmo que protected.	Mesmo que protected.

Métodos de visibilidade

Sendo utilizado o encapsulamento, tornando atributos ocultos, dá-se o lugar para a criação de métodos especiais para atribuir e recuperar valores de atributos de uma classe:

Getters: Os métodos "get" servem para retornar valores do estado interno do objeto. Possuem a nomenclatura get+Atributo(), sendo seu retorno o tipo compatível com o atributo em questão;

Setters: Os métodos "set" são utilizados para "colocar" valores no estado interno do objeto. Possuem a mesma ideia de nomenclatura que os métodos get, mas seu retorno é void. A sua assinatura refere-se a uma variável que será validada para que seu valor seja atribuído no estado interno do objeto.

Estes métodos são comummente públicos.

Um setter é exigido quando para o atributo há uma validação a ser garantida ou a alteração do atributo interfere no estado interno do objeto alterando novos atributos.

Um getter pode ser um retorno formatado ou um conjunto de informações do estado interno do objeto, não apenas o valor isolado do atributo.

Utilize chamadas a setters inclusive dentro da classe e de construtores, assim como getters.

Exemplo:

```
public class Pessoa {
            private String nome;  // Atributo encapsulado
            public Pessoa(String nome) {
                    setNome(nome);
            }
            public String getNome() {
                    return nome;
            }
            public void setNome(String nome) { //validador
                    if (nome.isEmpty())
                        this.nome = "Sem Nome";
                     else
                        this.nome = nome;
            }
        }
```

Modificadores especiais

Há uma série de especificadores especiais e que podem ser aplicados para classes, atributos e métodos combinados com especificadores de visibilidade.

Especificador	Classe	Atributo	Método
static	Implica numa classe que não precisa ser instanciada pra uso.	Implica no compartilhamento do valor para com todos os objetos instanciados da classe.	Implica na utilização do método pela própria classe, sem necessidade de instância.
abstract	A classe está incompleta, útil pra herança.	Não se aplica.	Método sem corpo. Requer escrita na descendência. Útil pra herança.
final	A classe é estéril, não pode gerar classes filhas. Útil pra herança.	O atributo é uma constante.	O método é único, não pode ser ocultado e nem sobrecarregado (Polimorfismo).

Os modificadores podem ser combinados entre si, o compilador avisará sobre conflitos ou impossibilidade.

Overload: Sobrecarga de métodos (pré assunto de Polimorfismo)

Os métodos podem ser sobrecarregados, **overload**. Reescritos mais do que uma vez, desde que sigam alguns critérios:

- Mantenha o tipo de retorno;
- Mantenha o mesmo nome do método;
- Diferenciem em assinatura.

Abaixo um exemplo de sobrecargas, sendo exemplos válidos e inválido.

```
class Classe {
    public   void       umMetodo   (double x) {...}
    public   void       umMetodo   (int x)    {...} // é
    public   double     umMetodo   (double x) {...} // não é
}
```

Override: Sobreposição de métodos (pré assunto de Herança e Polimorfismo)

A sobreposição de métodos, **override**, se dá pela reescrita de um método herdado.

Toda classe é descendente implicitamente da classe Object. Na classe Object do Java encontramos os seguintes métodos:

- public final Class<?> **getClass**() : Retorna a classe em tempo de execução do objeto;
- public int **hashCode**() : Retorna um valor numérico (hash) usado em coleções como HashMap e HashSet;
- public boolean **equals**(Object obj) : Compara se o objeto atual é "igual" ao outro (por padrão, compara referências);
- protected Object **clone**() throws CloneNotSupportedException : Cria e retorna uma cópia (clone) do objeto. Precisa implementar a interface Cloneable;
- public String **toString**() : Retorna uma representação em texto do objeto (por padrão: NomeDaClasse@hashcodeHex).

Desta forma, conhecendo a classe Object, podemos chamar métodos desta classe e sobrescrever os mesmos. Exemplo: toString();

```
//@Override
public String toString() {
       //super.toString(); //se quiser chamar o método da classe pai
           return "Aluno{" +
              "nome='" + nome + '\'' +
              ", idade=" + idade +
              ", matrícula='" + matricula + '\'' +
              '}';
}
```

Dentro de um método sobreposto, pode-se chamar o método original através de super, na mesma ideia do this. Sendo **super.metodo()** a chamada ao método na classe pai.