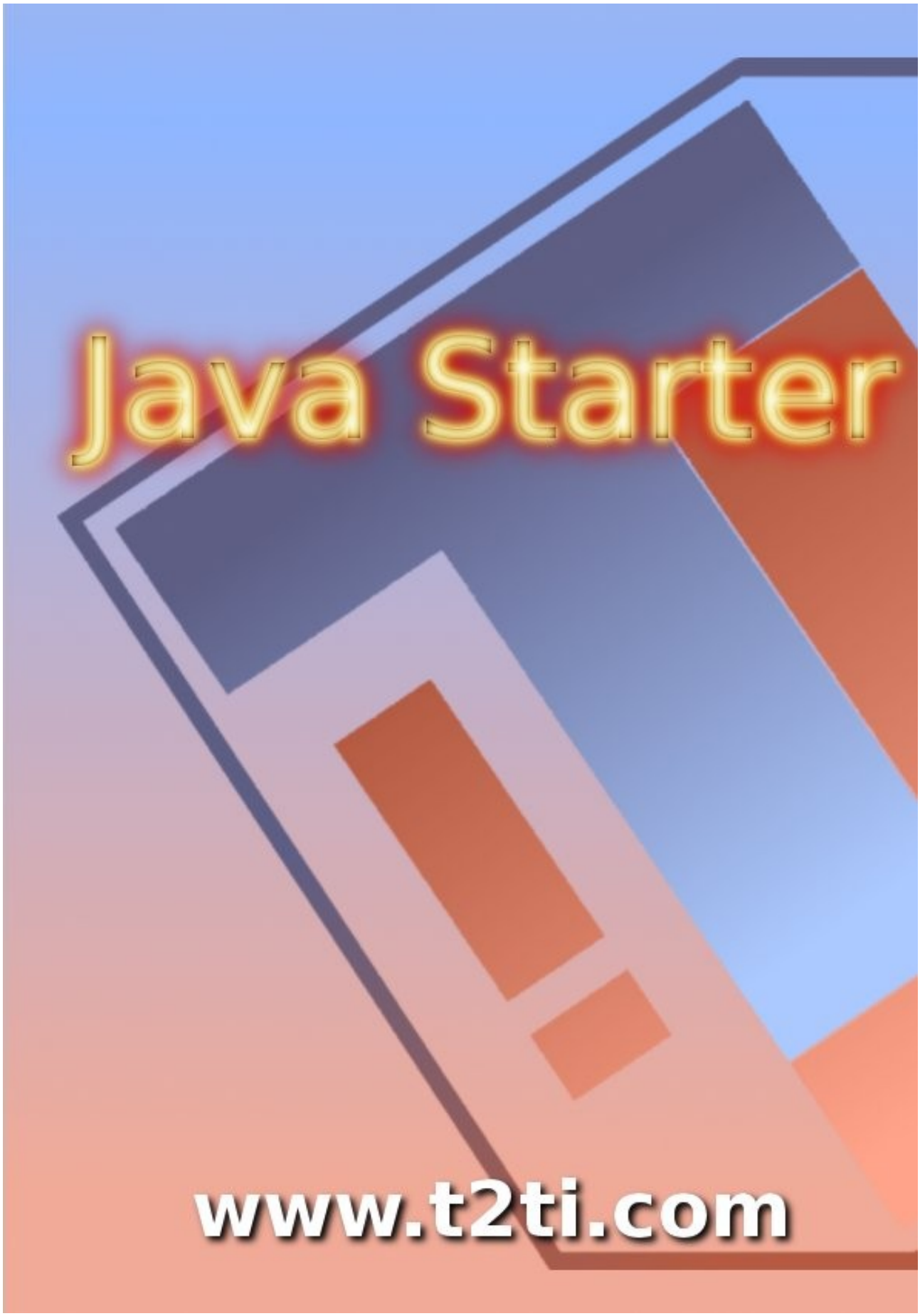
Java Starter
www.t2ti.com

**Apresentação**

O Curso Java Starter foi projetado com o objetivo de ajudar àquelas pessoas que têm uma base de lógica de programação e desejam entrar no mercado de trabalho sabendo Java,

A estrutura do curso é formada por módulos em PDF e por mini-cursos em vídeo. O aluno deve baixar esse material e estudá-lo. Deve realizar os exercícios propostos. Todas as dúvidas devem ser enviadas para a lista de discussão que está disponível para inscrição na página do Curso Java Starter no site www.t2ti.com. As dúvidas serão respondidas pelos instrutores Albert Eije, Cláudio de Barros e Miguel Kojiio, além dos demais participantes da lista.

Nosso objetivo é que após o estudo do Curso Java Starter o aluno não tenha dificuldades para acompanhar um curso avançado onde poderá aprender a desenvolver aplicativos para Web, utilizando tecnologias como Servlets e JSP e frameworks como Struts e JSF, além do desenvolvimento para dispositivos móveis.

Albert Eije trabalha com informática desde 1993. Durante esse período já trabalhou com várias linguagens de programação: Clipper, PHP, Delphi, C, Java, etc. Atualmente mantém o site www.alberteije.com.

Cláudio de Barros é Tecnólogo em Processamento de Dados.

Miguel Kojiio é bacharel em Sistemas de Informação, profissional certificado Java (SCJP 1.5).

O curso Java Starter surgiu da idéia dos três amigos que trabalham juntos em uma instituição financeira de grande porte.

Módulo 04

# Orientação a Objetos

## Classes, Objetos e Encapsulamento

## Introdução

Apesar de não termos sido apresentados formalmente, já estamos utilizando orientação a objetos nos programas que fizemos até agora, pois algumas das classes e tipos que utilizamos são verdadeiramente objetos. É claro que não estamos fazendo isto da forma correta ou melhor forma.

Mas porque "Programação Orientada a Objetos" ao invés da tradicional Programação Estruturada?

Antes de conhecermos a resposta devemos compreender que em algum momento o código Orientado a Objetos utiliza-se do paradigma Estruturado, a grande diferença é a forma como a aplicação é idealizada.

Um paradigma de programação, seja ele Estruturado ou Orientado a Objetos é a forma como a solução para um determinado problema é desenvolvida. Por exemplo, em Orientação a Objetos os problemas são resolvidos pensando-se em interações entre diferentes objetos, já no paradigma Estruturado procura-se resolver os problemas decompondo-os em funções e dados que somados formarão um programa.

Retornando a nossa pergunta, durante muito tempo a programação Estruturada foi o paradigma mais difundido porém, a medida que os programas foram tornando-se mais complexos, surgiu a necessidade de resolver os problemas de uma maneira diferente. Neste contexto surge o paradigma da Programação Orientada a Objetos.

Em Orientação a Objetos os dados e as operações que serão realizadas sobre estes formam um conjunto único (objeto), e a resolução de um problema é dada em termos de interações realizadas entre estes objetos.

Dizemos que um objeto encapsula a lógica de negócios, concentrando a responsabilidade em pontos únicos do sistema, viabilizando um maior reuso do código pela modularidade desta abordagem.

Benefícios da abordagem orientada a objetos:

- **Modularidade:** Uma vez criado um objeto pode ser passado por todo o sistema;
- **Encapsulamento:** Detalhes de implementação ficam ocultos externamente ao objeto;
- **Reuso:** Uma vez criado um objeto pode ser utilizado em outros programas;
- **Manutenibilidade:** Manutenção é realizada em pontos específicos do seu programa (objetos).

## Classes e Objetos

Objetos são "coisas" que temos no mundo real e abstraímos no mundo virtual para que possamos manipulá-los na resolução de problemas. Um objeto no mundo real sempre possui estado e comportamento, isto é, ele possui características e ações que são pertinentes a sua natureza. Para clarificar nada melhor do que alguns exemplos:

| Objeto | Estado | Comportamento |
|---|---|---|
| Pessoa | Nome, idade, RG | Falar, andar, cumprimentar |
| Cachorro | Nome, raça | Latir, correr |
| Conta bancária | Saldo, agência, número | Creditar, debitar |
| Carro | Cor, marca, modelo | Acelerar, frear, abastecer |

Observe que estado e comportamento, respectivamente, são transformados em **dados** e **procedimentos** quando programamos de forma estruturada e **atributos** e **métodos** quando utilizamos orientação a objetos.

Alguém pode perguntar: Legal, mas eu sou uma pessoa e possuo mais atributos (estado) do que os que foram apresentados! Cadê o restante (peso, altura e etc.)? Certamente, todos os objetos apresentados possuem estado e comportamento muito mais complexos do que os apresentados, no entanto, iremos criar um modelo para apenas aquilo o que nos interessa, ou melhor, interessa a resolução do problema. Seu programa não deve ter nada mais do que o necessário, fuja dos "Sistemas

Mundo" - no meu trabalho utilizamos esta expressão para identificar sistemas cujo escopo não está bem definido (limitado) – que são intermináveis.

Objetos são oriundos (instâncias) das classes. Uma **classe** é uma especificação para um determinado tipo de objeto, isto é, para que o objeto seja de determinada classe ele, obrigatoriamente, terá que respeitar a especificação. Por exemplo, vamos especificar que todo documento deve possuir, ao menos, foto, código, nome e data de nascimento.

Agora vamos **criar** um documento para o Alfredo e um para a Juliana:

| Documento | Documento 1 | Documento 2 |
|---|---|---|
| **Foto:** | Img1.png | Img4.png |
| **Código:** | 123456 | 789012 |
| **Nome:** | Alfredo | Juliana |
| **Data de nascimento:** | 20/05/1990 | 30/09/1987 |

Perceba que, nesta tabela a coluna **Documento** define a classe (especificação) enqüanto as colunas **Documento 1** e **Documento 2** são os objetos desta forma, cada documento particular terá um valor diferente para os atributos definidos na especificação. A compreensão desta diferença, entre **classes** e **objetos,** é parte fundamental do paradigma orientado a objetos.

Esta especificação em Java poderia ser da seguinte forma:

```
class Documento {

	//Estado
	String foto; //Nome do arquivo de imagem
	String nome; //Nome da pessoa
	Integer codigo; //Codigo deste documento
	String dataNascimento; //Data de nascimento

}
```

Perceba que nós acabamos de definir um novo tipo, Documento, e especificamos que para um objeto ser deste tipo ele deve necessariamente ter os atributos foto, nome, código e data de nascimento. Agora vamos implementar um programa que crie o documento do Alfredo (Documento 1 - tabela):

```
class Programa {
	public static void main(String[] args) {
```

```
                //Declarando meu objeto documento1
                Documento documento1;

                //Criando objeto documento1
                documento1 = new Documento();

                //Atribuindo os valores para o documento1
                documento1.codigo = 123456;
                documento1.nome = "Alfredo";
                documento1.foto = "Img1.png";
                documento1.dataNascimento = "20/05/1990";

                System.out.println("Código do documento: "+documento1.codigo);
        }
}
```

A algoritmo apresentado pode ser lido da seguinte forma: "Declaramos uma variável do tipo Documento denominada documento1, atribuímos a variável documento1 uma instância da classe Documento e para cada atributo do nosso objeto documento1 atribuímos os respectivos valores. Ao final o valor do código é impresso"

Perceba que os atributos do documento1 são acessados utilizando "." (ponto) sobre o nome da variável (documento1).

## Objetos na memória

Observe o seguinte programa:

```
class Programa {
        public static void main(String[] args) {
                Documento doc1 = new Documento();
                doc1.nome = "Alfredo";

                Documento doc2 = new Documento();
                doc2.nome = "Juliana";

                System.out.println("doc1: "+doc1.nome);
                System.out.println("doc2: "+doc2.nome);
        }
}
```

O resultado da execução deste programa imprime:

```
doc1: Alfredo
doc2: Juliana
```

Na memória o que acontece é o seguinte:

**doc1 = 253**

**doc2 = 260**

| | |
|---|---|
| 253 | **Documento**<br>nome = Alfredo |
| 254 | |
| 255 | |
| 256 | |
| 257 | |
| 258 | |
| 259 | |
| 260 | **Documento**<br>nome = Juliana |
| 261 | |

As minhas variáveis doc1 e doc2 não contém objetos, na verdade elas

indicam a posição (endereço) onde o objeto encontra-se na memória, e o Java procura nestes endereços os objetos. Os endereços são obtidos através da palavra reservada **new** que cria o novo objeto, aloca-o na memória e retorna o endereço de acesso.

Agora observe este outro programa muito similar:

```
class Programa {
        public static void main(String[] args) {
                Documento doc1 = new Documento();
                doc1.nome = "Alfredo";

                Documento doc2 = doc1;
                doc2.nome = "Juliana";

                System.out.println("doc1: "+doc1.nome);
                System.out.println("doc2: "+doc2.nome);
        }
}
```

Perceba que agora eu não estou mais criando um novo doc2, estou passando a referência (endereço) armazenado no meu doc1. Veja o resultado desta execução:

```
doc1: Juliana
doc2: Juliana
```

Desta vez o que aconteceu na memória foi o seguinte:

**doc1 = 253**

**doc2 = 253**

| | |
|---|---|
| 253 | **Documento**<br>nome = Juliana |
| 254 | |
| 255 | |
| 256 | |
| 257 | |
| 258 | |
| 259 | |
| 260 | |
| 261 | |

A variável doc1 tem como referência um objeto do tipo Documento armazenado na posição 253 da memória. A esta referência é atribuído o valor "Alfredo" ao campo nome, a seguir é declarada uma nova variável (doc2) cuja referência atribuída é a mesma de doc1, ou seja, o endereço 253, logo após, utilizando a variável de referência doc2, modificamos o valor do campo nome para "Juliana".

O resultado disto é que modificamos o campo nome sempre no mesmo objeto e no qual as duas variáveis (doc1 e doc2) estão referenciando. Devemos compreender que apesar das variáveis serem diferentes o valor contido nelas

(endereço) é o mesmo, logo o objeto que estas variáveis fazem referência também será o mesmo.

## Métodos

Ok, dentro de um objeto temos seu "estado" definido pelo conjunto de atributos. Mas cadê o comportamento? As ações que são realizadas sobre os atributos são os **métodos**, o conjunto de métodos define o **comportamento** de um objeto.

Imagine que agora o sistema a ser modelado é o de uma corrida de carros. O principal objeto de uma corrida de carros são os próprios carros – nenhuma novidade até aqui – para os quais nós devemos criar uma especificação (classe) para ser utilizada neste sistema imaginário.

O nosso carro de corrida terá seu estado definido pelo conjunto de atributos número de identificação, velocidade atual e velocidade máxima. O comportamento será definido pelo conjunto de métodos acelerar, frear, ligar e desligar.

Vamos criar a nossa classe apenas com a definição do estado, inicialmente:

```
class CarroCorrida {

        //Estado
        Integer numeroIdentificacao;
        Double velocidadeAtual;
        Double velocidadeMaxima;

        //Comportamento...
}
```

Pronto! Agora temos o nosso carro de corrida que não faz nada por enqüanto. Vamos adicionar os primeiros comportamentos, ligar e desligar. Estas ações (ligar e desligar) não retornam nenhum tipo de informação, o carro apenas emite um som, sendo assim, estes comportamentos podem ser implementados da seguinte forma:

```
        void ligar()
        {
                System.out.println("VRUUUMmmmmmmmmm");
        }


            void desligar()
        {
                System.out.println("MMMmmmm......");
        }
```

Observe que os dois métodos não retornam nenhuma informação para quem os aciona (invoca) isto é informado através da palavra reservada **void**.

Agora vamos implementar o método acelerar:

```
void acelerar()
{
	velocidadeAtual += 10;
	if(velocidadeAtual > velocidadeMaxima)
	{
		velocidadeAtual = velocidadeMaxima;
	}
}
```

Em acelerar() nós temos a primeira regra de negócio, cada acionamento deste comportamento adiciona 10 unidades de velocidade a nossa velocidade atual até o máximo permitido (atributo velocidade máxima), se a velocidade atual após a aceleração extrapolar o valor da velocidade máxima do carro então a velocidade atual será igual a velocidade máxima.

Vamos implementar o método frear:

```
void frear(Integer intensidadeFreada)
{
	if(intensidadeFreada > 100)
	{
		intensidadeFreada = 100;
	}else if(intensidadeFreada < 0)
	{
		intensidadeFreada = 0;
	}

	velocidadeAtual -= intensidadeFreada*0.25;

	if(velocidadeAtual < 0)
	{
		velocidadeAtual = 0.0;
	}
}
```

O método frear recebe um parâmetro que significa a intensidade com que o pedal de freio foi acionado. Esta intensidade pode ser um valor entre 0 e 100, a velocidade após a freada é o resultado da intensidade da freada multiplicada pelo fator 0.25 tudo isto diminuído da velocidade atual, caso o resultado desta operação seja menor do que 0 (zero) então o valor final será zero.

A nossa classe após a adição destes comportamento ficou da seguinte forma:

```
class CarroCorrida {

	//Estado
	Integer numeroIdentificacao;
	Double velocidadeAtual;
	Double velocidadeMaxima;

	//Comportamento...

	void ligar()
```

```
	{
		System.out.println("VRUUUMmmmmmmmmm");
	}

	void desligar()
	{
		System.out.println("MMMmmmm......");
	}

	void acelerar()
	{
		velocidadeAtual += 10;
		if(velocidadeAtual > velocidadeMaxima)
		{
			velocidadeAtual = velocidadeMaxima;
		}
	}

	void frear(Integer intensidadeFreada)
	{
		if(intensidadeFreada > 100)
		{
			intensidadeFreada = 100;
		}else if(intensidadeFreada < 0)
		{
			intensidadeFreada = 0;
		}

		velocidadeAtual -= intensidadeFreada*0.25;

		if(velocidadeAtual < 0)
		{
			velocidadeAtual = 0.0;
		}
	}
}
```

Podemos melhorar um pouco mais a nossa classe atribuindo valores **default** para alguns atributos. Por exemplo, nós sabemos que a velocidade inicial de qualquer veículo é zero, sendo assim vamos definir o nosso atributo velocidadeAtual inicialmente como zero e ao mesmo tempo vamos definir que a velocidade máxima destes carros será, inicialmente, 100. Para isto basta declarar estes atributos como é mostrado:

```
public class CarroCorrida {

	//Estado
	Integer numeroIdentificacao;
	Double velocidadeAtual = 0.0;
	Double velocidadeMaxima = 100.0;

	//...
```

Desta forma toda vez que um carro for instanciado o valor destas variáveis estará previamente atribuído.

Mas, observando novamente o nosso modelo de carro de corrida percebemos que faltam algumas características importantes. No nosso caso o piloto com seus atributos são fundamentais para uma corrida de automóveis.

Isto é fácil de resolver, primeiro identificamos os atributos que desejamos e

adicionamos na nossa classe CarroCorrida. Estes atributos são nome, idade e habilidade. Nossa nova classe fica da seguinte forma:

```
class CarroCorrida {

	//Estado
	Integer numeroIdentificacao;
	Double velocidadeAtual = 0.0;
	Double velocidadeMaxima = 100.0;
	String nomePiloto;
	Integer idadePiloto;
	Integer habilidadePiloto;

	//...
```

Em uma análise mais refinada e crítica é possível perceber que os atributos nomePiloto, idadePiloto e habilidadePiloto não fazem parte de um carro de corrida, na verdade quem tem estes atributos é o piloto. Neste caso nós podemos mudar a implementação e adicionar mais uma classe ao nosso modelo. O novo modelo ficaria da seguinte forma:

```
class Piloto {

	String nome;
	Integer habilidade;
	Integer idade;

}

class CarroCorrida {

	//Estado
	Integer numeroIdentificacao;
	Double velocidadeAtual = 0.0;
	Double velocidadeMaxima = 100.0;
	Piloto piloto;

	//...
```

Agora a classe CarroCorrida possui um atributo do tipo Piloto ou seja, a classe Piloto compõe a classe CarroCorrida. Já que temos um piloto com uma determinada habilidade, podemos alterar o nosso método acelerar() de forma que a habilidade do piloto seja levada em conta quando aceleramos, ou seja, quanto mais habilidoso for o piloto maior será a aceleração que ele irá imprimir ao veículo.

O novo método acelerar será da seguinte forma:

```
	void acelerar()
	{
		velocidadeAtual += 10 + piloto.habilidade*0.1;
		if(velocidadeAtual > velocidadeMaxima)
		{
			velocidadeAtual = velocidadeMaxima;
		}
	}
```

Temos o nosso carro de corrida com piloto. Agora é chegada a hora de criarmos o nosso programa que simula uma corrida de carros entre duas equipes, a equipe "Velocidade" e a equipe "Trapaceiros":

```
class Corrida {

	public static void main(String[] args) {
		//Criacao dos carros que irao correr
		CarroCorrida carroEquipeVelocidade = new CarroCorrida();
		CarroCorrida carroEquipeTrapaceiros = new CarroCorrida();

		//Criacao dos pilotos de cada equipe
		Piloto pilotoEquipeVelocidade = new Piloto();
		Piloto pilotoEquipeTrapaceiros = new Piloto();

		//Atributos do piloto da equipe Velocidade
		pilotoEquipeVelocidade.nome = "Piloto 1";
		pilotoEquipeVelocidade.idade = 25;
		pilotoEquipeVelocidade.habilidade = 75;

		//Atributos do piloto da equipe Trapaceiros
		pilotoEquipeTrapaceiros.nome = "Piloto 2";
		pilotoEquipeTrapaceiros.idade = 27;
		pilotoEquipeTrapaceiros.habilidade = 65;

		//Os pilotos sao colocados nos seus carros
		carroEquipeVelocidade.piloto = pilotoEquipeVelocidade;
		carroEquipeTrapaceiros.piloto = pilotoEquipeTrapaceiros;

		//Identificao dos carros
		carroEquipeVelocidade.numeroIdentificacao = 1;
		carroEquipeTrapaceiros.numeroIdentificacao = 2;

		//Carros sao ligados
		carroEquipeVelocidade.ligar();
		carroEquipeTrapaceiros.ligar();

		//Inicia a corrida
		carroEquipeVelocidade.acelerar();
		carroEquipeTrapaceiros.acelerar();

		carroEquipeVelocidade.acelerar();
		carroEquipeTrapaceiros.acelerar();

		carroEquipeVelocidade.acelerar();
		carroEquipeTrapaceiros.velocidadeAtual = 200.0;

	}
}
```

Tudo estava acontecendo conforme planejado – ambos os carros foram ligados, partiram com velocidade zero e possuem a mesma velocidade máxima (100) – no entanto em um determinado momento, um dos carros corrompeu a integridade do meu programa de duas formas:

1. A velocidade atual ficou maior que a velocidade máxima do carro (padrão é 100);

2. Não respeitou o algoritmo de aceleração do veículo que permite o aumento de velocidade em intervalos de 10 unidades mais um adicional de acordo com a habilidade do piloto.

Diagnosticamos corretamente o problema e agora que o problema foi descoberto nós precisamos resolvê-lo, e para isto nós iremos entrar em um dos conceitos fundamentais do paradigma Orientado a objetos, o "encapsulamento".

## ENCAPSULAMENTO

Apesar do nome relativamente esquisito o conceito é simples. Encapsulamento refere-se ao isolamento entre as partes do programa. Uma das principais formas de fazer isso é proibindo o acesso direto as variáveis de um objeto por objetos externos.

Para limitar o acesso aos membros do objeto (métodos e atributos), utilizamos os modificadores de acesso existentes em java (**public, private, protected e default**).

Estes modificadores funcionam da seguinte forma:

- **public:** Qualquer objeto pode acessar o membro;
- **default:** Qualquer objeto do mesmo pacote pode acessar o membro e subclasses de outros pacotes;
- **protected:** O membro é acessível apenas por objetos do mesmo pacote;
- **private:** O membro é acessível apenas internamente (próprio objeto);

Os dois modificadores de acesso mais utilizados desta lista são o **public** e o **private,** e são com estes que nós iremos nos preocupar. Normalmente, os métodos são públicos (public) e os atributos private (privados), isto ocorre pois nós desejamos que os atributos de um objeto só possam ser alterados por ele mesmo, desta forma nós inviabilizamos situações imprevistas.

Vamos entender isto voltando ao nosso programa de corrida de carros. Observe que na linha identificada pela seta vermelha existe uma violação da integridade.

Em um determinado momento do programa um valor é atribuído de forma aleatória, sem respeitar a regra de negócio. Observe novamente o trecho de código a

seguir:

```
            ...
            carroEquipeVelocidade.acelerar();
            carroEquipeTrapaceiros.acelerar();

            carroEquipeVelocidade.acelerar();
            carroEquipeTrapaceiros.velocidadeAtual = 200.0;
    }...
```

Compreendeu o problema? A equipe trapaceiros simplesmente "apelou" e atribuiu o valor 200 ao campo velocidadeAtual, desconsiderando que a velocidade máxima do carro é 100 e sem respeitar o algoritmo (regra de negócio) implementado no método acelerar.

Para resolver isto nós iremos limitar o acesso aos atributos da classe CarroCorrida de forma que estes só possam ser acessado internamente. Observe como fica a nova classe:

```
class CarroCorrida {

    //Estado
    private Integer numeroIdentificacao;
    private Double velocidadeAtual = 0.0;
    private Double velocidadeMaxima = 100.0;
    private Piloto piloto;
```

Pronto! Agora os meus atributos só poderão ser acessados pela próprio objeto. Muito bem, mas criamos um outro problema... Como o meu programa irá saber a velocidade de cada carro e determinar quem está na frente ou atrás durante a corrida?

Vamos entrar em uma outra importante convenção quando programamos em Java. São os métodos **getters** e **setters**. Este métodos são responsáveis por fornecer meios modificarmos o "estado" - lembra-se? - de um objeto, isto é, meios de acessarmos e modificarmos valores dos atributos de um objeto.

Mas é claro que nós iremos criar estes métodos apenas se forem realmente necessários, ou seja, nós não damos acesso aos atributos que não interessam a outros objetos, ou seja, interessam apenas ao próprio objeto.

A convenção para estes métodos é a seguinte:

- **Getters**: Método que retorna o atributo, é sempre composto pela palavra get[nome do atributo]. Ex: getIdade(), getSalario().
- **Setters**: Método que atribui/modifica o valor de um atributo, é sempre composto pela palavra set[nome do atributo] e o parâmetro do mesmo tipo do atributo. Ex: setIdade(Integer idade), setSalario(Double salario).

A nossa classe com estes métodos fica da seguinte forma:

```
class CarroCorrida {

	//Estado
	private Integer numeroIdentificacao;
	private Double velocidadeAtual = 0.0;
	private Double velocidadeMaxima = 100.0;
	private Piloto piloto;

	//Comportamento...
	//Demais métodos foram suprimidos apenas para melhorar a visualização
	//...

	public Integer getNumeroIdentificacao() {
		return numeroIdentificacao;
	}

	public void setNumeroIdentificacao(Integer numeroIdentificacao) {
		this.numeroIdentificacao = numeroIdentificacao;
	}

	public Piloto getPiloto() {
		return piloto;
	}

	public void setPiloto(Piloto piloto) {
		this.piloto = piloto;
	}

	public Double getVelocidadeAtual() {
		return velocidadeAtual;
	}
}
```

Percebeu que eu não criei getter e setter para todos os atributos? Isto acontece pois não há necessidade de fornecer meios de acesso e/ou modificação a todos os atributos (observe o uso dos modificadores de acesso antes da declaração de cada variável e cada método – public e private).

Vamos observar atributo a atributo e enteder o pensamento Orientado a objetos:

- **numeroIdentificacao**: Este atributo deve ser acessado e atribuído/modificado (**getter** e **setter** foram criados) por qualquer objeto, pensando no mundo real (no caso a corrida que estou modelando) o número de identificação é variável, isto é, a direção do campeonato pode mudar aleatóriamente entre corridas, ou seja, é necessário fornecer meios de modificá-lo e acessá-lo a outros objetos;

- **velocidadeAtual**: Este atributo pode apenas ser acessado (**getter**), isto ocorre pois desejo que a velocidade dos meus carros de corrida sejam modificadas apenas através dos métodos **frear()** e **acelerar()**

respeitando os algoritmos implementados;

- **velocidadeMaxima**: Ninguém pode acessar ou modificar/atribuir valores para este atributo (sem **getter** ou **setter**) pois, no meu modelo de corrida de carros, a velocidade máxima de cada um é pré-definida e não será alterada. O carro já é criado sabendo que pode alcançar a velocidade máxima de 100 unidades de velocidade (Km/h, velocidade da luz e etc.), observe que o meu modelo de carro de corrida é assim para fins didáticos, mas normalmente nós iremos implementar o atributo velocidadeMaxima de forma que seja função de outras variáveis como motor, potência, chassis e etc;

- **piloto**: Este atributo pode ser acessado e modificado/atribuído (**getter** e **setter**) por outros objetos do programa. Pensando no modelo seria o seguinte: Nós podemos trocar de piloto do carro entre corridas, durante as corridas e ainda deixá-lo sem piloto quando estiver parado, logo eu preciso de meios para que isto seja possível.

Antes de continuar, vamos abordar um outro assunto o **Construtor.**

O construtor não é um método, é um bloco responsável pela criação da classe. Ele é executado toda a vez que utilizamos a palavra reservada **new**, e sua única função é criar uma instância (objeto) da classe. Quando o construtor não é explicitamente declarado o compilador insere o construtor default, isto é, um construtor sem parâmetros e com o corpo vazio.

No nosso caso, a classe CarroCorrida tem o seguinte construtor declarado implicitamente (default), ou seja, colocado pelo compilador:

```
public CarroCorrida()
{

}
```

Para mostrar uma implementação do construtor nós iremos definir que a velocidade maxima de um veiculo e o seu número de identificação são definidos durante a sua construção – através de dois parâmetros – portanto nós não iremos utilizar o construtor default. O nosso novo construtor fica da seguinte forma:

```
	public CarroCorrida(Integer numeroIdentificacao, Double velocidadeMaxima) {
		this.numeroIdentificacao = numeroIdentificacao;
		this.velocidadeMaxima = velocidadeMaxima;
	}
```

A palavra reservada **this**, no corpo do nosso novo construtor, significa que nós estamos acessando o membro, neste caso o atributo, do próprio objeto que está sendo instanciado.

Agora toda a vez que uma instância da classe CarroCorrida for criada necessariamente deverá ser definida a velocidade máxima do carro, desta forma garantimos que a velocidade máxima do veículo é atribuída no momento da sua criação. Portanto para criar um novo carro de corrida nós devemos proceder conforme trecho de código abaixo:

```
class Corrida {

	public static void main(String[] args) {
		//Criacao dos carros que irao correr
		CarroCorrida carroEquipeVelocidade = new CarroCorrida(1, 100.0);
		CarroCorrida carroEquipeTrapaceiros = new CarroCorrida(2, 100.0);
		//...
```

O mais interessante é que posso ter carros com diferentes velocidades máximas, definidas no momento da construção.

Observe que até agora nós nos preocupamos apenas com a classe CarroCorrida, mas a classe piloto também deve seguir ser modificada a fim de encapsularmos os seus atributos. O resultado é o seguinte:

```
class Piloto {

	private String nome;
	private Integer habilidade;
	private Integer idade;

	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public Integer getHabilidade() {
		return habilidade;
	}
	public void setHabilidade(Integer habilidade) {
		this.habilidade = habilidade;
	}
	public Integer getIdade() {
		return idade;
	}
	public void setIdade(Integer idade) {
		this.idade = idade;
	}

}
```

Ao aplicarmos os conceitos apresentados a nossa classe Corrida fica da seguinte forma:

```
class Corrida {

	public static void main(String[] args) {
		//Criacao dos carros que irao correr
		CarroCorrida carroEquipeVelocidade = new CarroCorrida(1, 100.0);
		CarroCorrida carroEquipeTrapaceiros = new CarroCorrida(2, 100.0);

		//Criacao dos pilotos de cada equipe
		Piloto pilotoEquipeVelocidade = new Piloto();
		Piloto pilotoEquipeTrapaceiros = new Piloto();

		//Atributos do piloto da equipe Velocidade
		pilotoEquipeVelocidade.setNome("Piloto 1");
		pilotoEquipeVelocidade.setIdade(25);
		pilotoEquipeVelocidade.setHabilidade(75);

		//Atributos do piloto da equipe Trapaceiros
		pilotoEquipeTrapaceiros.setNome("Piloto 2");
		pilotoEquipeTrapaceiros.setIdade(27);
		pilotoEquipeTrapaceiros.setHabilidade(65);

		//Os pilotos sao colocados nos seus carros
		carroEquipeVelocidade.setPiloto(pilotoEquipeVelocidade);
		carroEquipeTrapaceiros.setPiloto(pilotoEquipeTrapaceiros);

		//Identificao dos carros
		carroEquipeVelocidade.setNumeroIdentificacao(1);
		carroEquipeTrapaceiros.setNumeroIdentificacao(2);

		//Carros sao ligados
		carroEquipeVelocidade.ligar();
		carroEquipeTrapaceiros.ligar();

		//Inicia a corrida
		carroEquipeVelocidade.acelerar();
		carroEquipeTrapaceiros.acelerar();

		carroEquipeVelocidade.acelerar();
		carroEquipeTrapaceiros.acelerar();

		carroEquipeVelocidade.acelerar();
		carroEquipeTrapaceiros.acelerar();

	}
}
```

Preste atenção neste novo código e compare-o com o que foi implementado anteriormente, veja que agora não acessamos/modificamos nenhum atributo de forma direta, somos obrigados a utilizar os métodos de cada uma das classes.

## Exercícios

*Respeite o encapsulamento em todos os exercícios.

Aprenda com quem também está aprendendo, veja e compartilhe as suas respostas no nosso Fórum:

Exercícios – Módulo 04 – OO, Classes e Objetos, Modificadores de Acesso

1. Modifique a classe CarroCorrida, vista durante este módulo. Adicione um outro construtor que receba apenas um parâmetro (velocidade máxima).
2. Crie uma classe Motor com o atributo potência (inteiro que varia entre 1 e 100), implemente um construtor que receba o parâmetro potência e verifique se o valor encontra-se dentro dos limites estabelecidos (1 - 100). Caso o valor extrapole o limite superior ou inferior o valor da potência deve ser o valor do limite extrapolado.
3. Adicione a classe CarroCorrida o atributo motor (utilize classe Motor do exercício 2). Altere o método acelerar() de forma que ao resultado da aceleração já implementado sejam adicionados 10% do valor da potência do motor.
4. Escreva uma classe "Contador", que apresente métodos para informar o valor inicial, incrementar, decrementar e imprimir o valor atual.
5. Crie uma classe que represente um ponto no plano cartesiano, lembrando que um ponto no plano cartesiano é representado pelas coordenadas no eixo x e no eixo y.
6. Crie uma classe que represente um triângulo, utilize a classe desenvolvida no exercício anterior para identificar os vértices do triângulo.
7. Implemente uma classe Pessoa com os seguintes atributos: Nome, idade e CPF.
8. Utilizando a classe implementada no exercício anterior crie um programa que instancie 2 pessoas com todos os atributos e imprima os valores.
9. Implemente uma classe que represente uma sala de aula, esta sala pode ter o máximo de 10 alunos, se extrapolar este limite deve ser impressa uma mensagem avisando que o número máximo de alunos foi atingido.
10. Implemente uma classe de Endereço com os seguintes atributos: Estado, Cidade, Bairro, Rua, CEP e telefone.
11. Adicione a classe Pessoa desenvolvida no exercício 7 um atributo de endereço utilizando a classe desenvolvida no exercício 10.
12. Implemente um método construtor para a classe Pessoa de forma que uma instância desta classe seja criada apenas se possuir nome, idade e cpf.

**13.** Implemente uma classe que simule um cadastro de pessoal. Esta classe deve armazenar até 100 pessoas (utilize a classe Pessoa) com seus respectivos endereços. Esta classe deve ter os seguintes comportamentos: permitir o cadastramento e exclusão de pessoas do cadastro.

**14.** Faça uma classe Calculadora que realize as 4 operações matemáticas básicas (soma, divisão, multiplicação e subtração) sobre dois valores (double) passados como parâmetros e retorne o resultado. Crie um programa que realize as 4 operações e imprima os resultados obtidos.

**15.** Faça uma classe CalculadoraComercial, esta classe deve realizar, além das 4 operações básicas, o cálculo de porcentagens. O cálculo da porcentagem deve ser efetuado sobre 2 parâmetros o valor total (double) e a porcentagem a ser obtida (inteiro) retornando o resultado. Atenção, a classe CalculadoraComercial deve ter um atributo Calculadora e transferir a responsabilidade pela realização das 4 operações matemáticas básicas para o objeto Calculadora, ou seja, a única implementação nova será o cáculo da porcentagem.

**16.** Faca uma classe Conta que contenha o cliente (utilize a classe Pessoa desenvolvida nos exercícios anteriores), o numero da conta, o saldo e o limite. Estes valores deverao ser informados no construtor. Faça um método depositar e um método sacar. O método sacar irá devolver true ou false, em razão da disponibilidade ou não de saldo. Faça um método saldo que retorne o saldo do cliente.

**17.** Utilize a classe Pessoa e a classe Sala, desenvolvidas nos exercícios anteriores, e mais uma classe Escola com atributos nome, CNPJ e salas (máximo de 20 salas ocupadas). Faça um programa que:

- Crie uma escola;
- Adicione a esta escola algumas salas;
- Adicione as salas pessoas (alunos);
- Transfira um aluno de uma sala para outra.