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 07

Orientação a Objetos

Herança, Sobreescrita e Polimorfismo

Introdução

Em relação ao paradigma Orientado a Objetos nós já vimos o que é uma Classe, aprendemos também que um objeto nada mais é do que a instanciação de uma classe e que um dos pilares da Orientação a Objetos é o encapsulamento.

Neste módulo e no próximo nós iremos continuar nossos estudos relacionados aos conceitos fundamentais da programação Orientada a Objetos. A seguir estudaremos Herança, Sobreescrita e Polimorfismo.

Herança

No paradigma orientado a objetos podemos definir um conjunto de classes em uma estrutura hierárquica onde cada uma das classes "herda" características da suas superiores nesta estrutura.

No mundo real temos que uma pessoa herda características de seus pais que herdaram de seus avós e assim por diante. Respeitando as limitações das linguagens de programação, o que acontece quando utilizamos herança no nosso código é uma tentativa de implementar esta propriedade da natureza.

Para ficar mais claros vamos partir para um exemplo. Imagine que estivéssemos implementando um programa que fosse utilizado em um concessionária de veículos. Esta concessionária vende carros de passeio, jipes e veículos utilitários. Para implementar as classes destes veículos poderíamos criar três classes distintas,

A seguir a classe Jipe:

```
public class Jipe {

	private String marca;
	private Double capacidadeTanqueCombustivel;

	public Double getTanqueCombustivel()
	{
		return capacidadeTanqueCombustivel;
```

```
	}
	public void setTanqueCombustivel(Double capacidadeTanqueCombustivel)
	{
		this.capacidadeTanqueCombustivel = capacidadeTanqueCombustivel;
	}
	public String getMarca() {
		return marca;
	}
	public void setMarca(String marca) {
		this.marca = marca;
	}

	public void acelerar()
	{
		//Código aqui
	}
	public void frear()
	{
		//Código aqui
	}

	public void ligarTracao4x4()
	{
		//Código aqui
	}
}
```

A classe que representa o nosso carro de passeio:

```
public class CarroPasseio {

	private String marca;
	private Double capacidadeTanqueCombustivel;

	public Double getTanqueCombustivel()
	{
		return capacidadeTanqueCombustivel;
	}
	public void setTanqueCombustivel(Double capacidadeTanqueCombustivel)
	{
		this.capacidadeTanqueCombustivel = capacidadeTanqueCombustivel;
	}
	public String getMarca()
	{
		return marca;
	}
	public void setMarca(String marca)
	{
		this.marca = marca;
	}

	public void acelerar()
	{
		//Código aqui
	}
	public void frear()
	{
		//Código aqui
	}

	public void ligarArCondicionado()
	{
		//Código aqui
	}
}
```

E, finalmente, a classe que representa o nosso veículo utilitário:

```
public class Utilitario {

	private String marca;
	private Double capacidadeTanqueCombustivel;
```

```
        public Double getTanqueCombustivel()
        {
                return capacidadeTanqueCombustivel;
        }
        public void setTanqueCombustivel(Double capacidadeTanqueCombustivel)
        {
                this.capacidadeTanqueCombustivel = capacidadeTanqueCombustivel;
        }
        public String getMarca()
        {
                return marca;
        }
        public void setMarca(String marca)
        {
                this.marca = marca;
        }

        public void acelerar()
        {
                //Código aqui
        }
        public void frear()
        {
                //Código aqui
        }

        public void ligarFarolMilha()
        {
                //Código aqui
        }
}
```

Perceba que, nas linhas destacadas em amarelo, temos as características que dizem respeito a apenas um dos tipos de veículos, isto é, no nosso modelo, apenas jipes têm **tração 4x4**, somente os carros de passeio têm opção de **ligar o ar condicionado** e **farol de milha** é uma característica existente apenas nos veículos utilitários

Todas os demais estados e comportamentos – você ainda lembra o que são estado e comportamento de um objeto? – são idênticos, na verdade para criar estas três classes foi realizado uma simples cópia e cola dos membros que são iguais. Isto é válido, por enqüanto, afinal de contas estas características permanecerão idênticas para os três tipos (Jipe, Carro de Passeio e Utilitário).

Agora vamos explorar um pouco mais o nosso modelo, a nossa concessionária vai começar a vender outros 5 tipos de veículos diferentes, cada um com suas peculiaridades, lá vamos nós criar mais 5 classes e copiar e colar mais 5 vezes! E mais, o cliente do software identificou uma característica importante dos veículos que deve ser adicionada ao modelo, o **código do chassi**. Lá vamos nós de novo implementar mais 8 – afinal são 5 novos veículos mais os 3 anteriores – atributos com getter e setter para contemplar esta característica.

Na verdade nada disto seria necessário se nós tivéssemos utilizado corretamente o conceito de herança da programação Orientada a Objetos. Perceba que com exceção das características que estão grifadas em amarelo todas as demais

são comuns a todos os veículos, ou seja, é possível que tenhamos uma **superclasse** (classe Pai) Veiculo e que todas as demais herdem estas características comuns dela.

A nossa classe Veiculo ficaria da seguinte forma:

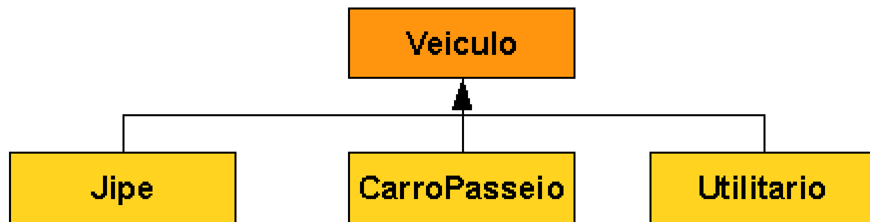
```
public class Veiculo {

	private String marca;
	private Double capacidadeTanqueCombustivel;


	public Double getTanqueCombustivel() {
		return capacidadeTanqueCombustivel;
	}

	public void setTanqueCombustivel(Double capacidadeTanqueCombustivel) {
		this.capacidadeTanqueCombustivel = capacidadeTanqueCombustivel;
	}

	public String getMarca() {
		return marca;
	}

	public void setMarca(String marca) {
		this.marca = marca;
	}

	public void acelerar() {
		//Código aqui
	}

	public void frear() {
		//Código aqui
	}

}
```

Tudo o que era comum a todos os veículos foi concentrado na classe Veiculo. Para que as demais classes **herdem** estas características nós devemos utilizar a palavra reservada **extends**.

As novas classes têm como **superclasse** a classe Veiculo:

```
public class Jipe extends Veiculo{

	public void ligarTracao4x4()
	{
		//Código aqui
	}
}

public class CarroPasseio extends Veiculo{

	public void ligarArCondicionado()
	{
		//Código aqui
	}
}

public class Utilitario extends Veiculo {

	public void ligarFarolMilha()
	{
		//Código aqui
	}
}
```

Graficamente podemos representar esta hierarquia de classes da seguinte forma:



Nesta nova hierarquia de classes toda a subclasse de Veiculo irá compartilhar as suas características, dizemos que um jipe, carro de passeio ou utilitário são todos do tipo Veiculo. E da mesma forma toda a subclasse de Jipe irá herdar as características tanto de Veiculo quanto de Jipe.

O exemplo abaixo mostra como, apesar de não ser declarado especificamente na classe Jipe, posso acessar através do getter o atributo **capacidadeTanqueCombustivel**:

```
Jipe jipe = new Jipe();
Double capacidadeTanqueJipe = jipe.getCapacidadeTanqueCombustivel();
jipe.setCapacidadeTanqueCombustivel(48.0);
```

Percebeu!? No exemplo acima estou manipulando o atributo através dos métodos (getter e setter) sem tê-los declarados explicitamente na classe Jipe pois, o compilador sabe que na verdade estou **herdando** estas características da classe Veiculo.

Esta hierarquia de classes garante que todas as mudanças efetuadas na classe Veiculo se propagam para as suas **subclasses**. Por exemplo, se for solicitado adicionar o comportamento "buzinar", e este comportamento for uma características que todos os veículos devem ter, então basta adicioná-lo a classe Veiculo que automaticamente todas as **subclasses** compartilharão a mesma funcionalidade.

Abaixo temos o exemplo desta modificação na classe Veiculo:

```
public class Veiculo {

	private String marca;
	private Double capacidadeTanqueCombustivel;
```

```
		//Getters, setters e demais métodos aqui...

	public void buzinar()
	{
		System.out.println("Béééémmmm");
	}

}
```

E agora o meu Jipe também buzina:

```
		Jipe jipe = new Jipe();
		Double capacidadeTanqueJipe = jipe.getCapacidadeTanqueCombustivel();
		jipe.setCapacidadeTanqueCombustivel(48.0);
		jipe.buzinar();
```

Desta forma nós evitamos aquele super-trabalho de copiar, colar e manter várias classes cada vez que uma alteração é solicitada, concentramos tudo o que é possível e pertinente na nossa classe Veiculo e realizamos a manutenção do código em apenas um ponto da aplicação.

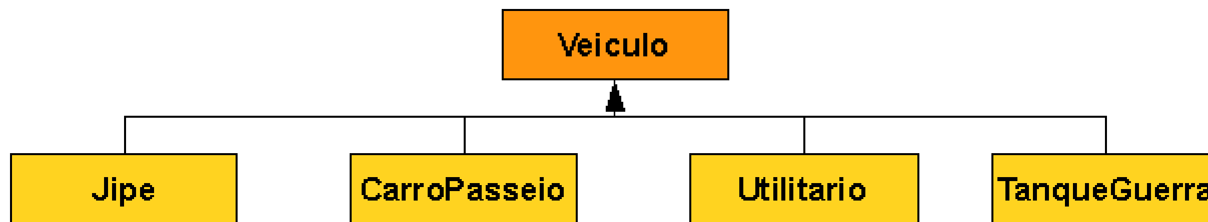
Assim, a herança viabiliza o **reuso** do código existente pois, como podemos perceber o código da classe Veiculo é reutilizado por todas as suas subclasses e, ainda mais, podemos ter uma hierarquia de herança de tamanho indefinido onde todas as subclasses irão sempre herdar características das superclasses até a raiz da estrutura hierárquica (classe do topo da hierarquia).

Vamos brincar um pouco com o nosso modelo, imagine que fossêmos vender o nosso software de gerência de concessionárias para uma concessionária localizada em um país que esteja em uma zona conflituosa e, além destes veículos a concessionária vende também “tanques de guerra” - agora eu apelei! :) - portanto devemos adicionar a nossa hierarquia mais uma classe.

A classe que representa um tanque de guerra fica da seguinte forma:

```
public class TanqueGuerra extends Veiculo {

	public void atirarMetralhadora()
	{
		//Código aqui
	}

	public void atirarCanhao()
	{
		//Código aqui
	}
}
```

Graficamente a nova hierarquia de classes é a seguinte:



E novamente eu sei que a minha classe TanqueGuerra também possui as características herdadas da classe Veiculo, ou seja, além de atirar com a metralhadora e o canhão – características próprias deste tipo de veículo – o nosso "tanque de guerra" poderá buzinar, acelerar, frear e etc.

Abaixo temos um exemplo do uso da classe TanqueGuerra:

```
TanqueGuerra osorio = new TanqueGuerra();
osorio.atirarCanhao();//derivado da classe TanqueGuerra
osorio.buzinar();//derivado da classe Veiculo
```

Agora nós podemos pensar o seguinte, o som emitido pela buzina ("Béééémmmm") fica bem para um Jipe, Utilitário e até quem sabe um Tanque de Guerra, mas nunca uma senhora de idade iria adquirir um carro de passeio com uma buzina deste tipo.

Sendo assim nós precisamos modificar o som da buzina apenas para os carros de passeio de forma a deixá-la mais condizente com a realidade do uso deste veículo.

Para isto devemos conhecer mais um dos fundamentos da Orientação a Objetos a **sobreescrita de métodos**.

Sobreescrita (override)

A sobreescrita de métodos é uma característica que permite alteração do comportamento de uma **superclasse** pelas suas **subclasses**. Ao **sobreescrevermos** um método nós estamos fornecendo uma nova implementação para o mesmo comportamento.

Para exemplificar vamos alterar o comportamento da nossa buzina para que seja mais adeqüada ao um carro de passeio. A classe Veiculo permanece do mesmo jeito, modificamos apenas a classe CarroPasseio conforme abaixo:

```java
public class CarroPasseio extends Veiculo{

	public void ligarArCondicionado()
	{
		//Código aqui
	}

	public void buzinar()//Estamos sobreescrevendo o método buzinar() da superclasse - Veiculo
	{
		System.out.println("Fon fon");
	}

}
```

E ao executarmos o main abaixo:

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

		TanqueGuerra osorio = new TanqueGuerra();
		osorio.buzinar();
		CarroPasseio carroMadame = new CarroPasseio();
		carroMadame.buzinar();

	}
```

Obtemos a seguinte saída no console:

TanqueGuerra

CarroPasseio

```
Béééééémmmm
Fon fon
```

Perceba que o “som” emitido pelo tanque de guerra continuou o mesmo, derivado da classe Veiculo, porém o som emitido pelo carro de passeio mudou, ou seja, alteramos o comportamento pré-definido para o método buzinar().

Nesta situação o método buzinar da classe Veiculo é completamente ignorado e substituído pelo da classe CarroPasseio, porém pode haver situação que seja interessante a manutenção da implementação existente com adição de implementação própria.

Para que isto seja possível nós devemos, ao sobreescrever o método, adicionar a palavra reservada **super** ao corpo do método que estamos fornecendo nova implementação e invocar o mesmo método da superclasse.

No exemplo abaixo utilizamos a implementação do método buzinar da superclasse e ao mesmo tempo adicionamos nova implementação:

```java
public class TanqueGuerra extends Veiculo {

	public void atirarMetralhadora()
	{
		//Código aqui
	}

	public void atirarCanhao()
	{
		//Código aqui
```

```
	}

	public void buzinar()
	{
		super.buzinar();
		System.out.println("BÉÉÉÉÉMMMMMMM");
	}
}
```

A palavra reservada **super** é uma referência a superclasse. Ao executarmos este método a JVM irá primeiro invocar e executar o método da superclasse para depois continuar a execução local.

Ao sobreescrevermos um método o modificador de acesso – neste caso **public** – nunca pode ser mais restritivo do que o do método que está sendo sobreescrito, ou seja, não podemos alterar o modificador de acesso de forma que o método tenha sua visibilidade reduzida, por exemplo, se tentarmos mudar a assinatura do método buzinar, conforme abaixo, o compilador não irá permitir:

```
protected void buzinar()
```

No entanto o inverso é válido, podemos aumentar a visibilidade de um método, ou seja, se um método for declarado **protected** e aumentarmos a visibilidade (**public**) dele ao sobreescrevermos não teremos problemas com o compilador.

Sobreescrever é diferente de sobrecarregar (overload)

Sobreescrever é diferente de sobrecarregar um método. Um método só é sobreescrito se a **assinatura** do mesmo é mantida intacta, isto é, o tipo de retorno e os parâmetros (inclusive a seqüência) não são alterados.

Observe a classe CarroPasseio:

```
public class CarroPasseio extends Veiculo{

	public void ligarArCondicionado()
	{
		//Código aqui
	}

	public void buzinar(String som)
	{
		System.out.println(som);
	}

}
```

Observe o que acontecerá ao executarmos o código abaixo:

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

		CarroPasseio carroMadame = new CarroPasseio();
		carroMadame.buzinar();
		carroMadame.buzinar("Fon! Fon!");

	}
```

O resultado desta execução é o seguinte:

```
Béééééémmmm
Fon! Fon!
```

Observe que o nosso novo método (grifado em amarelo) está sobrecarregando o método já existente, na verdade temos dois métodos buzinar, com e sem parâmetros, respectivamente, o método da classe CarroPasseio e o método derivado (herdado) da classe Veiculo.

A sobrecarga de método cria um método novo na classe mantendo o método existente.

Polimorfismo

O polimorfismo, que significa muitas formas, permite que uma mesma referência se comporte de formas diferentes. Utilizando corretamente o polimorfismo nós podemos fazer o código ficar mais claro, genérico e, portanto, reutilizável.

Segue abaixo um exemplo:

```java
public class MainModulo09 {

	/**
	 * @param args
	 */
	public static void main(String[] args) {

		Veiculo veiculo = criarVeiculo(args[0]);
		veiculo.buzinar();

	}

	public static Veiculo criarVeiculo(String tipo)
	{
		if(tipo == null)
		{
			return new CarroPasseio();
		}else if(tipo.equals("utilitario"))
		{
			return new Utilitario();
		}else if(tipo.equals("tanque"))
		{
			return new TanqueGuerra();
		}else if(tipo.equals("passeio"))
		{
			return new CarroPasseio();
```

```
            }
            return new Utilitario();
        }

}
```

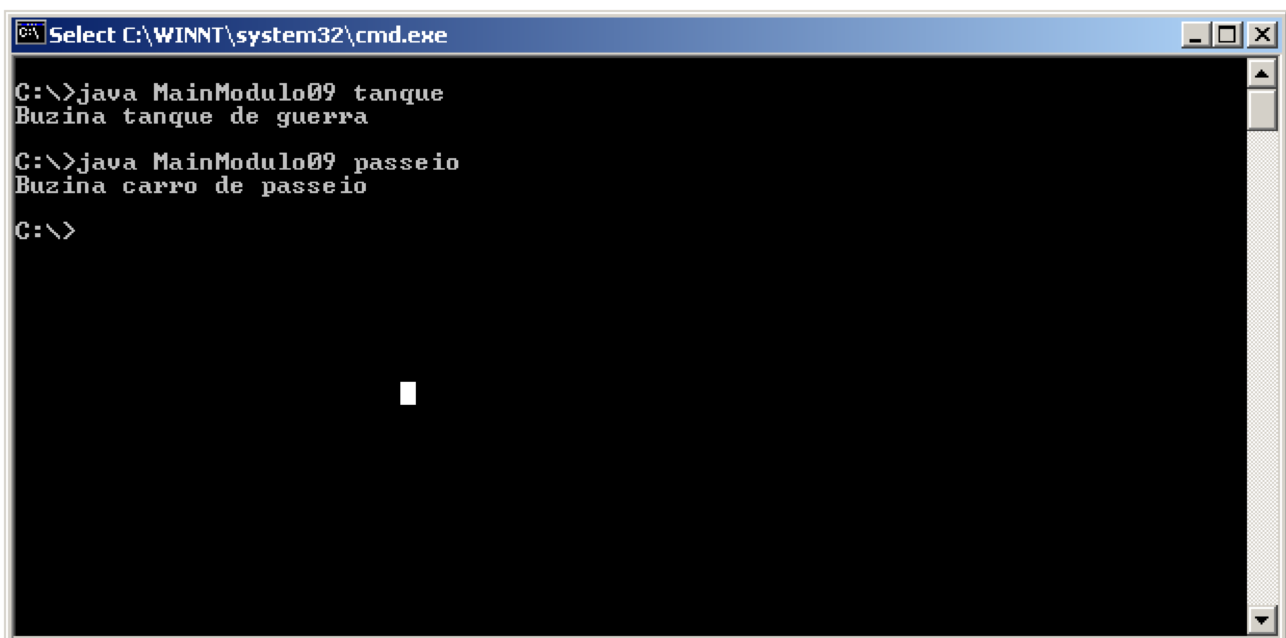
A invocação do método buzinar() na variável "veiculo" é feita de forma polimórfica, isto é, pode possuir diversos comportamentos diferentes. O resultado desta invocação será apenas conhecido em tempo de execução pois, dependendo do parâmetro de entrada do método main o objeto retornado pelo método criarVeiculo() será de tipos diferentes.

Perceba que a variável "veiculo" é do tipo Veiculo, ou seja, ela pode referenciar objetos das classes Jipe, CarroPasseio, TanqueGuerra e Utilitario, afinal, conforme visto, todas estas classes são também do tipo Veiculo (Herança).

A JVM só conhecerá a referência da variável "veiculo" em tempo de execução, que será dependente do parâmetro de entrada do método main, ou seja, a JVM até sabe que em tempo de execução esta variável irá referenciar um objeto do tipo Veiculo, ou uma de suas subclasses, porém ela não sabe qual dos tipos de Veiculo será (Jipe, TanqueGuerra, Utilitario ou CarroPasseio).

Durante a execução da aplicação a JVM irá identificar qual tipo a variável está referenciando e assim irá invocar a implementação do método buzinar() adequada.

Observe abaixo a execução do código:

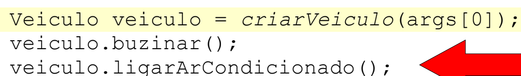


Veja que, conforme explicado, durante cada execução do programa a JVM

identificou a correta implementação do método buzinar() neste caso respectivamente as implementações da classe TanqueGuerra e CarroPasseio.

Se, por acaso, o método buzinar() não tivesse sido sobreescrito em nenhuma das subclasses (TanqueGuerra e CarroPasseio) a JVM então iria invocar a implementação existente na classe Veiculo.

Ao declararmos a variável "veiculo" utilizamos o tipo Veiculo, superclasse das demais classes, observe o trecho de código abaixo:



```
Veiculo veiculo = criarVeiculo(args[0]);
veiculo.buzinar();
veiculo.ligarArCondicionado();
```

A situação acima identificada resulta em erro de compilação, isto acontece porque o método "ligarArCondicionado()" não faz parte da classe Veiculo ou de uma de suas **superclasses**, ou seja, o compilador verifica se o método existe em tempo de compilação.

Desta forma, mesmo que você tenha certeza que naquele momento a variável fará referência um objeto do tipo CarroPasseio, o compilador não irá aceitar este tipo de construção. No entanto existem soluções para isto, a primeira e mais simples é substituir o tipo da variável, conforme abaixo:

```
CarroPasseio veiculo = criarVeiculo(args[0]);
veiculo.buzinar();
veiculo.ligarArCondicionado();
```

A segunda forma é executando um cast, a idéia é a mesma do cast de tipos primitivos porém, neste caso, nós avisamos ao compilador que a variável faz referência a uma das subclasses da hierarquia de classes, observe a construção abaixo:

```
Veiculo veiculo = criarVeiculo(args[0]);
veiculo.buzinar();
((CarroPasseio)veiculo).ligarArCondicionado();
```

Podemos dizer que o compilador entende algo como: "Opa?! Beleza! Se você está garantindo que eu vou encontrar um CarroPasseio neste endereço então eu deixo você invocar este método porém espero que saiba o que está fazendo pois se, em tempo de execução, a referência for de outro tipo a JVM irá emitir uma exceção bem espalhafatosa".

Modificador final

O modificador **final** quando utilizado em classes ou métodos **bloqueia**, respectivamente, a extensão e a sobreescrita, isto é, utilizamos este modificador quando não desejamos que nossa classe tenha subclasses ou o método seja sobreescrito.

Devido às suas características este modificador deve ser utilizado com parcimônia e sabedoria pois ele vai contra um dos pilares da orientação a objetos: Reuso de código através da herança e/ou sobreescrita.

Abaixo segue um exemplo do uso:

```
public final class Jipe extends Veiculo{

	public void ligarTracao4x4()
	{
		//Código aqui
	}

	public void buzinar()
	{
		System.out.println("Buzina jipe");
	}
}
```

Conforme dito anteriormente, ao adicionar o modificador **final** a declaração da classe estou inviabilizando qualquer tipo de estensão por outras classes, isto é, apesar da classe Jipe estender a classe Veiculo, nenhuma outra classe poderá estender, ser subclasse, da classe Jipe.

Em métodos a aplicação deste modificador é feita da seguinte forma:

```
public class Jipe extends Veiculo{

	public void ligarTracao4x4()
	{
		//Código aqui
	}

	public final void buzinar()
	{
		System.out.println("Buzina jipe");
	}
}
```

Agora a classe Jipe pode ser estendida porém qualquer uma de suas subclasses não poderá sobreescrever o método buzinar(). Isto é interessante quando desejamos que determinado comportamento não seja alterado em hipótese nenhuma.

Nos pacotes básicos da linguagem Java existem diversas classes que são modificadas pelo **final**, temos como exemplo a classe String e a classe Math.

Se foi dito que o modificador final vai contra os pilares da Orientação a Objetos porque as próprias classes desenvolvidas pelos gurus da SUN utilizam este modificador?

A resposta é a seguinte, quando estamos manipulando uma String nós devemos ter certeza que o comportamento esperado será encontrado, isto é, ao contrário de outras classes para a classe String nós esperamos um comportamento estável e consistente ao longo do tempo. Imagine os problemas decorrentes do uso de uma String que se comporta de maneira diferente da esperada?!

Da mesma forma com a classe Math, imagine os problemas decorrentes da realização de uma operação matemática em que o resultado obtido seja diferente do esperado pois, um outro programador estendeu a classe e resolveu modificar as implementações – e por acaso ele não sabia tanto de matemática quanto achava que sabia :).

Nestes casos o modificador **final** é utilizado a fim de evitar um mal maior aos programas em detrimento ao melhor uso do paradigma orientado a objetos.

Quando você for utilizar, se for utilizar, o modificador **final** pense sempre nas implicações e nos objetivos que espera obter com o seu uso.

Classe Object

Para encerrar este módulo vejamos a classe Object. Em Java todas as classes, eu disse todas as classes! String, Math, as minhas classes, as suas classes e etc. estendem a classe Object e portanto são do tipo Object também.

Desta forma mesmo quando declaramos uma classe conforme abaixo:

```
public class Classe {
}
```

Implicitamente o compilador interpreta esta classe como:

```
public class Classe extends Object{
}
```

Por isso todos os objetos – objetos e não tipos primitivos! – sempre irão passar (**true**) por um teste conforme abaixo:

```java
if(qualquerObjeto instanceof Object)
{
        //Sempre entrará neste bloco
}
```

Lê-se: "Todo objeto é instância da classe Object".

A seguir temos os alguns dos principais métodos da classe Object:

Método	Descrição
equals()	Indica quando um objeto é igual a este
toString()	Retorna a representação textual deste objeto
hashCode()	Retorna o código hash para este objeto

Todos estes métodos podem ser sobreescritos, e muitas vezes o são, os métodos equals() e hashCode serão abordados no módulo de Coleções. Abaixo temos um exemplo da utilização do método toString():

```java
public class Jipe extends Veiculo{

        public void ligarTracao4x4()
        {
                //Código aqui
        }

        public final void buzinar()
        {
                System.out.println("Buzina jipe");
        }

        public String toString()
        {
                return "toString do Jipe";
        }
}
```

Observe que o método toString(), da classe Object, está sendo sobreescrito pela minha classe Jipe, a seguir temos trecho de código que imprime um objeto do tipo Jipe:

```java
Veiculo veiculo = new Jipe();
System.out.println(veiculo);
```

O resultado deste código é o seguinte:

```
toString do Jipe
```

Quando imprimimos um objeto, a JVM invoca o método toString() do objeto, se este método não foi sobreescrito então o método toString() invocado é o da

classe Object.

Se não tivesse sido sobreescrito um dos possíveis resultados a ser obtido seria este:

```
Jipe@10385c1
```

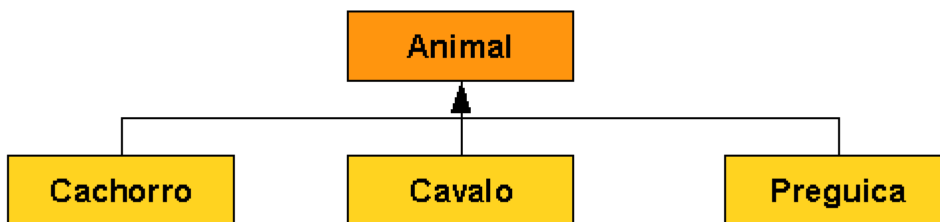
O que é uma "representação textual" significativa do objeto para a JVM, mas não para nós seres humanos :).

EXERCÍCIOS

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

Exercícios – Módulo 07 – Herança, Sobreescrita e Polimorfismo

1 Crie uma hierarquia de classes conforme abaixo com os seguintes atributos e comportamentos (observe a tabela), utilize os seus conhecimentos e distribua as características de forma que tudo o que for comum a todos os animais fique na classe Animal:



Cachorro	Cavalo	Preguica
Possui Nome	Possui Nome	Possui Nome
Possui Idade	Possui Idade	Possui Idade
Deve emitir som	Deve emitir som	Deve emitir som
Deve correr	Deve correr	Deve subir em árvores

2 Implemente um programa que crie os 3 tipos de animais definidos no exercício anterior e invoque o método que emite o som de cada um de forma polimórfica, isto é, independente do tipo de animal.

3 Implemente uma classe Veterinario que contenha um método examinar() cujo parâmetro de entrada é um Animal, quando o animal for examinado ele deve emitir um som, passe os 3 animais com parâmetro.

4 Crie uma classe Zoologico, com 10 jaulas (utilize um array) coloque em cada jaula um animal diferente, percorra cada jaula e emita o som e, se o tipo de animal possuir o comportamento, faça-o correr.

5 Resolva a seguinte situação utilizando os conceitos aprendidos. Uma empresa quer manter o registro da vida acadêmica de todos os funcionários, o modelo deve contemplar o registro das seguintes informações, de forma incremental:

- **5.1** Para o funcionário que não estudou, apenas o nome e o código funcional;
- **5.2** Para o funcionário que concluiu o ensino básico, a escola;
- **5.3** Para o funcionário que concluiu o ensino médio, a escola;
- **5.4** Para o funcionário que concluiu a graduação, a Universidade;

6 Estenda o modelo implementado no exercício anterior de forma que todo funcionário possua uma renda básica de R$ 1000,00 e:

- **6.1** Com a conclusão do ensino básico a renda total é renda básica acrescentada em 10%;
- **6.2** Com a conclusão do ensino médio a renda total é a renda do nível anterior acrescentada em 50%;
- **6.3** Com a conclusão da graduação a renda total é a renda do nível anterior acrescentada em 100%;
- **6.4** Todos os cálculos são efetuados sempre sobre a última renda obtida.

7 Crie um programa que simule uma empresa com 10 funcionários, utilize um array, sendo que a escolaridade dos funcionários é distribuída da seguinte forma: 40% ensino básico, 40% ensino médio e 20% nível superior. Calcule os custos da empresa com salários totais e por nível de escolaridade, utilize a classe funcionário desenvolvida no exercício anterior.

8 Faça uma hierarquia de Comissões, crie as comissões de Gerente, Vendedor e Supervisor. Cada uma das comissões fornece um adicional ao salário conforme abaixo:

- **8.1** Gerente: R$1500,00
- **8.2** Supervisor: R$600,00
- **8.3** Vendedor: R$250,00

9 Adicione a classe funcionário um atributo referente as comissões desenvolvidas no exercício anterior. Corrija o método renda total de forma que ele some ao

valor da renda calculada o adicional da comissão do funcionário.

10 Refaça o exercício 7 considerando que 10% dos funcionários são Gerentes, 20% são supervisores e 70% são vendedores.

11 Sobreescreva o método toString de forma que ele imprima o nome do funcionário, a comissão e o salário total. Imprima todos os funcionários da empresa criada no exercício 7.