CENTRO UNIVERSITÁRIO DE PATOS DE MINAS
DISCIPLINA: DESENVOLVIMENTO WEB II
EDUARDO HENRIQUE SILVA

PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA

SUMÁRIO

1 INTRODUÇÃO

Um carro é montado de várias partes e componentes, assim como, chassi, portas, motor, rodas, sistema de frenagem. Vários componentes utilizados para a montagem de um carro são reutilizáveis, por exemplo, as rodas podem ser usadas em vários carros com a mesma especificação. E no desenvolvimento de software? Você pode desenvolver um sistema selecionado um componente aqui, outro lá, e esperar que ele execute?

A resposta é “Sim”, a programação orientada a objetos (POO) é baseada no conceito de classes e objetos que são utilizadas para modelar entidades do mundo real. A POO permite um alto nível de abstração para resolver problemas do mundo real. Por exemplo, para desenvolver um jogo de corrida você deverá modelar a sua aplicação de acordo com os “objetos reais” que aparecem em seu jogo, como:

- Carro: velocidade máxima;
- Piloto: nome, altura, peso;
- Pista: distância.

O mais importante é que algumas dessas classes “objetos reais” como Piloto e Carro poderiam ser reutilizadas em outras aplicações como, software de autoescola, com ou sem alterações.

2 CLASSES E OBJETOS

Em Java, uma classe é uma definição de objetos do mesmo tipo. Em outras palavras, uma classe é uma especificação que define e descreve características e comportamentos comuns a todos os objetos do mesmo tipo.

Um objeto é uma instância de uma classe. Todas as instâncias de uma classe têm propriedades similares, como descritas na definição da classe. Por exemplo, podemos definir uma classe denominada “Carro” e criar duas instâncias (objetos) da classe “Carro”, como, “Gol 1.0”, “Gol 1.6”. A Figura 1 apresenta um exemplo real e especificação/instância.

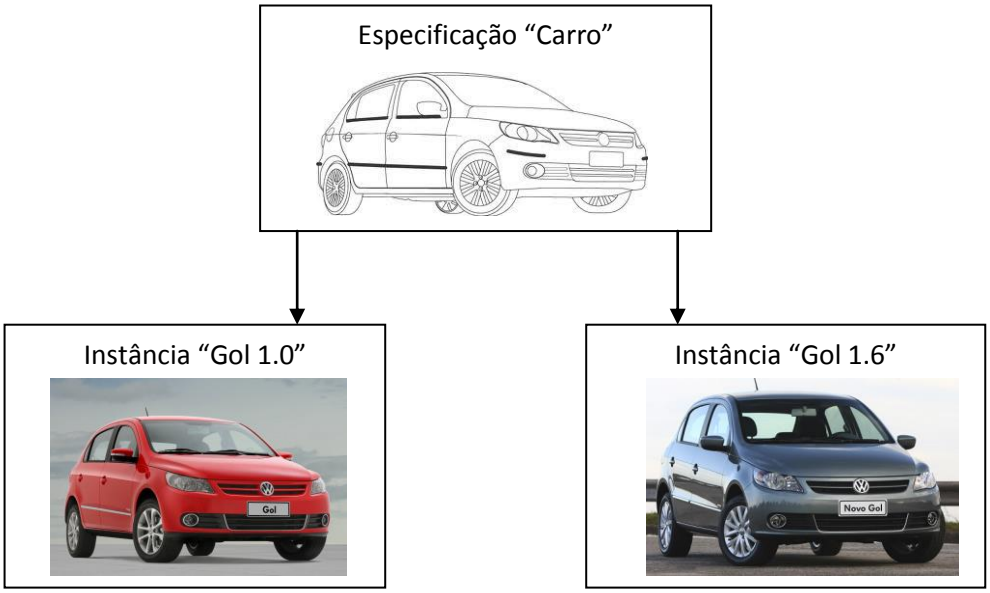

Figura 1 - Exemplo real de Especificação - Instância

Em uma montadora, o engenheiro cria o modelo, ou melhor, a especificação que define as características e comportamentos de seus carros. A partir dessa especificação são criadas as instâncias. Cada instância possui suas próprias características e comportamentos. Na POO não é diferente, a nossa especificação se dá através da definição de classes e nossos objetos são as instâncias criadas a partir de uma classe. Para definir uma classe em Java, utilizamos a palavra-chave **class**:

```
class Carro { // Nome da classe
}
```

Agora, a partir dessa classe podemos criar os objetos “Gol 1.0”, “Gol 1.6”. Para instanciar um objeto utilizamos a palavra chave **new**. Ao instanciar uma classe, ela é alocada na memória principal.

```
Carro gol10 = new Carro();// Objeto “Gol 1.0”
Carro gol16 = new Carro();// Objeto “Gol 1.6”
```

Na criação os objetos acima, foram necessários a definição das referências. Uma referência de objeto possui o endereço de memória que a instância está alocada, é através dessa referência que podemos acessar nossos objetos.

```
Carro gol10; //Referência de objeto denominada gol10
```

Ao criar uma referência de objeto, primeiramente devemos informar qual o tipo do objeto que será armazenado nessa referência, o exemplo acima, criou uma referência do tipo "Carro". Após a definição do tipo é necessário informarmos um nome, para posteriormente acessarmos esse objeto através de seu nome. A referência acima ainda não foi inicializada, ou seja, ainda não instanciamos nenhum objeto e atribuímos a ela. Quando uma referência não é inicializada, ela assume o valor **null**.

Agora, para iniciar uma referência de objeto deveremos criar uma instância do objeto, e atribuí-lo a referência, lembrado para criar um objeto utilizamos a palavra chave **new**.

```
Carro gol10;
gol10 =new Carro();
```

2.1 CARACTERÍSTICAS E COMPORTAMENTOS

Um objeto possui características e comportamentos, fazendo uma abstração com o mundo real, um carro possui características como: cor, peso, potência, dentre outros. E também possui comportamentos como: acelerar, frenar, acionar airbag, dentre outros.

Na POO as características de um objeto são definidas pelas variáveis de instância. Os valores atribuídos para as variáveis de instância de um objeto compõe o estado do objeto. O comportamento de um objeto é definido pelos métodos. Os métodos são onde a lógica da classe está armazenada, onde os algoritmos são executados e dados manipulados.

O código abaixo define uma classe com as características (variáveis) cor e velocidade e com os comportamentos (métodos) acelerar e frenar.

```
public class Carro {

	String cor; //Variável de instância
	double velocidade; //Variável de instância

	void acelerar(){ //Método

	}

	void frenar(){//Método

	}
}
```

2.2 O OPERADOR PONTO (.)

O operador "." é usado para referenciar um membro de um objeto (variável e método). Através dele podemos invocar métodos e acessar os dados das variáveis presentes nos objetos. O código abaixo apresenta a criação do objeto "`Carro`" e a referencia de seus membros. Foi utilizado o operador "." para acessar os membros do objeto.

```
Carro carro10 = new Carro();//Cria um instância para a referência carro10

carro10.cor = "Vermelho"; //Acessa a variável cor e altera o seu valor
carro10.velocidade = 100; //Acessa a variável velocidade e altera o seu valor

carro10.acelerar();//Invoca o método acelerar
carro10.frenar();//Invoca o método frenar

System.out.println(carro10.cor); //Imprime o valor armazenado na variável cor
System.out.println(carro10.velocidade); //Imprime o valor armazenado na variável
velocidade
```

Se você tentar acessar algum membro de instância sem inicializar a referência, será lançada uma exceção *NullPointException*, dizendo que a sua referência não está apontando para nenhum objeto na memória.

```
Carro carro10; ";//"Carro carro10;" = "Carro carro10 = null;"
carro10.cor = "Vermelho";//Nesse momento será lançada uma exceção]
```

2.3 CONSTRUTORES

Um construtor é um método especial que possui o mesmo nome da classe em que ele pertence. Um construtor é utilizado para inicializar as variáveis em um objeto.

Um método construtor é diferente de um método comum, pelos seguintes aspectos:

- O nome do construtor deve ser o mesmo nome da classe, a primeira letra sempre maiúscula;
- O construtor não possui retorno (ou implicitamente retorna *void*);
- O construtor pode somente ser invocado pelo comando `new`. Ele pode somente ser usado uma vez para inicializar a instância;

- O construtor não é herdado (será explicado posteriormente).

Toda classe que não define nenhum método construtor, implicitamente possui um construtor sem argumentos por padrão. Na classe abaixo não foi definido nenhum método construtor.

```
public class Carro {

}
```

Porém, implicitamente essa classe possui um construtor sem argumentos. Como apresenta a classe abaixo.

```
public class Carro {

	Carro(){ // Método construtor
		//Seu código
	}
}
```

Ao criar um objeto, você precisa usar a palavra chave **new**, seguido da chamada de um método construtor. Pelo código "`new Carro();`" você está invocando o método construtor "Carro() { }".

Como dito anteriormente, podemos inicializar as variáveis de um objeto, para isso precisamos definir um método construtor em nossa classe. Veja o código.

```
public class Carro {

	String cor;
	double velocidade;

	Carro(String cor, double velocidade){ // Método construtor
		this.cor = cor;
		this.velocidade = velocidade;
	}

}
```

Quando definimos um construtor na classe, o construtor padrão (sem argumentos) não é definido explicitamente. Então não podemos chamar "`new Carro();`". O correto seria:

```
Carro carro16= new Carro("Vermelho", 100); // Método construtor definido na classe
```

Podemos definir mais de um construtor? A resposta é sim. Podemos ter vários métodos construtores em uma classe, desde que as assinaturas dos métodos sejam diferentes. O código abaixo define três métodos construtores.

A assinatura de método é dada pelo número e tipo dos argumentos do método.

```
public class Carro {

	String cor;
	double velocidade;


	public Carro() {
		//Seu código
	}

	public Carro(String cor) {
		this.cor = cor;
	}

	Carro(String cor, double velocidade){ // Método construtor
		this.cor = cor;
		this.velocidade = velocidade;
	}

}
```

Poderíamos criar uma instância dessa classe invocando qualquer um dos três construtores. Por exemplo: "`new Carro();`"."`new Carro("Vermelho", 100);`"."`new Carro("Vermelho");`".

2.3.1 A palavra chave "*this*"

Você pode utilizar a palavra chave "***this***" para referenciar a "**essa**" instância dentro de uma definição de classe. Um dos principais usos desse comando é para resolver a ambiguidade.

```
public class Carro {

	String cor; // Variável chamada "cor"

	void alterarCor(String cor){ // Método com um argumento também chamado "cor"
		this.cor = cor; // Realiza a atribuição da valor presente na variável "cor" passada no argumento para a variável de instância "cor"
	}

}
```

O código "`this.cor`" referência a variável de instância "cor", enquanto a variável "cor" é referente ao argumento do método.

Pelo "`this`" você pode invocar qualquer membro da instância (métodos e variáveis) desde que elas não sejam estáticas. Posteriormente falaremos de membros estáticos.

3 MODIFICADORES

A Tabela 1 apresenta alguns modificadores Java, seguido dos elementos em que os modificadores podem ser empregados.

Modificador	Classe	Método	Variável de instância	Variável Local
default	Sim	Sim	Sim	Não
public	Sim	Sim	Sim	Não
protected	Não (apenas classes internas)	Sim	Sim	Não
private	Não (apenas classes internas)	Sim	Sim	Não
static	Não (apenas classes internas)	Sim	Sim	Não
final	Sim	Sim	Sim	Sim
abstract	Sim	Sim	Não	Não

Tabela 1 - Modificadores Java

3.1 MODIFICADORES DE ACESSO E ENCAPSULAMENTO

Java é uma linguagem centrada em pacotes, ao desenvolver as suas classes deverá coloca-las dentro de pacotes, para uma melhor organização. Os pacotes também são usados para agrupar um conjunto de classes relacionadas, além, de permitir a restrição de acesso de classes e membros através do uso dos modificadores de acesso. Essa restrição também é conhecida como encapsulamento, onde a lógica de uma classe fica oculta aos usuários da classe. Os usuários usam os serviços da classe, mas, sem saber como isso ocorre internamente.

Um modificador de acesso define o nível de visibilidade de uma classe, método, variável e construtor. Temos quatro modificadores de acesso.

3.1.1 Modificador *default*

O modificador *default* é utilizado quando nós não declaramos explicitamente qualquer outro modificador de acesso. Qualquer elemento sem modificador de acesso assume o *default*. O elemento com esse modificador apenas ficará disponível para classes que estão no mesmo pacote e pela classe que ela foi declarada. No código abaixo, não especificamos nenhum modificador de acesso, então apenas classes do pacote "`br.aula.oo`" poderão acessar a classe, a variável é o método.

```
package br.aula.oo;
class Carro {
	String cor;
	void acelerar(){
	}
}
```

3.1.2 Modificador *public*

A classe, método ou variável com o modificador *public* pode ser acessado por qualquer outra classe e pela classe que ela foi declarada. O código abaixo apresenta membros com o acesso *public*.

```
package br.aula.oo;
public class Carro {
	public String cor;
	public void acelerar(){
	}
}
```

3.1.3 Modificador *private*

Os métodos e variáveis declarados com o modificador *private*, somente poderão ser acessados dentro da própria classe onde eles foram declarados. Ou seja, nenhuma classe poderá acessar os membros privados.

```
package br.aula.oo;

public class Carro {

	private String cor;

	private void acelerar(){

	}
}
```

E a classe, porque não pode ser privada? Não faz sentido você criar um *protected* a classe e ninguém poder usa-la. Então as classes podem ser criadas apenas com os modificadores *default* e *public*, com exceção das classes internas que podem ser *private*.

3.1.4 Modificador *protected*

Os métodos e variáveis declarados com o modificador *protected*, poderão ser acessados pela classe que ela foi declarada e por subclasses (subclasses são classes que herdam características e comportamentos de uma classe pai, usando a palavra chave **extends**, entraremos em detalhes posteriormente). No Java membros com *protected*, poderão também ser acessados por classes que estão no mesmo pacote em que o membro foi definido. Então, subclasses, classes no mesmo e a própria classes podem ver membros *protected*.

```
package br.aula.oo;

public class Carro {

	protected String cor;

	protected void acelerar(){

	}
}
```

Igual ao modificador *private*, não podemos ter classes *protected*, a menos que ela seja uma classe interna.

3.1.5 Métodos *get*/*set*

Normalmente, o conteúdo das variáveis é acessado por métodos *get* e *set*, derivados do conceito do encapsulamento. O código abaixo apresenta a classe “Carro” onde encapsulamos a variável cor, definindo o seu conteúdo como *private*. Por esse motivo nenhuma classe é capaz de acessar essa variável e alterar o seu valor. Porém, a classe “Carro” pode acessar essa variável, então foi definido dois métodos públicos, “setCor(String cor)” responsável por alterar o valor da variável “cor” e o método “getCor(String cor)” responsável por retornar o valor da variável “cor”.

```
package br.aula.oo;

public class Carro {

	private String cor;

	public String getCor() {
		return cor;
	}

	public void setCor(String cor) {
		this.cor = cor;
	}

}
```

Esse conceito permite que suas classes fiquem menos acopladas, pois você restringe o acesso direto a variável, e passa essa responsabilidade para os métodos. Esse conceito é altamente recomendado em um projeto orientado a objetos.

3.2 MODIFICADORES DE NÃO ACESSO

3.2.1 Modificador *static*

Membros estáticos (*static*) estão relacionados com classes e não com objetos. Ou seja, podemos utilizar métodos e variáveis estáticas sem termos uma instância de objeto. Um membro *static* possui o mesmo valor para todos os objetos, pois, elas não são membros de instância e sim membros de classe. O código abaixo apresenta uma classe com dois métodos estáticos.

```
package br.aula.oo;

public class Carro {

	public static double capacidadeCombustivel;

	public static void ligar(){

	}

}
```

Mas como podemos acessar os membros sem ter uma instância?

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

	Carro.capacidadeCombustivel = 10;

	Carro.ligar();

	System.out.println(Carro.capacidadeCombustivel);

}
```

Estamos acessando membros públicos, lembre-se sempre de verificar a visibilidade dos membros. Um membro *private static* é apenas visível na própria classe.

O código mostra como podemos acessar membros estáticos sem ter uma instância. Para acessar membros estáticos verifique primeiramente a visibilidade, e depois chame "Nome_Da_Clase.Nome_Do_Membro". Mas podemos acessar membros estáticos com objetos? A resposta é sim.

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

	Carro.capacidadeCombustivel = 10; //Acesso ao membro estático pela classe

	Carro carro10 = new Carro();//Criação de uma instância

	System.out.println(carro10.capacidadeCombustivel); //Acesso ao membro
estático pelo objeto. Será impresso o valor "10", pois a variável é de classe uma
vez alterada é alterada para todas as instâncias dessa classe.

}
```

É importante saber que um método estático apenas pode invocar variáveis e métodos estáticos. O código abaixo não compila, pois o método estático "ligar" está chamando uma variável não estática.

```
public double capacidadeCombustivel;

public static void ligar(){
	capacidadeCombustivel = 10;
}
```

O código abaixo em bem similar ao anterior, porém, a variável não estática está sendo invocada através de uma instância, esse código compila.

```
public double capacidadeCombustivel;

public static void ligar(){
	new Carro().capacidadeCombustivel = 10;
}
```

- Métodos estáticos não podem invocar membros não estáticos de sua classe, porém, eles podem invocar membros não estáticos de instâncias;
- Métodos de instâncias podem acessar membros estáticos;
- Métodos estáticos podem acessar membros estáticos.

3.2.2 Modificador *final*

Quando utilizamos a palavra chave *final* em uma classe, estamos dizendo que aquela classe não pode ser subclasse, ou seja, não pode ser herdada. A classe *String* da API Java é uma classe *final*. Um dos princípios da POO é reusabilidade, porque, eu restringiria uma classe para não ser herdada? Você marcará uma classe como final apenas quando você quiser garantias de que ninguém irá sobrescrever os seus métodos (alterar o comportamento).

```
public final class Carro {//Classe com o modificador final

}


public class Sedan extends Carro{ //Não compila, nenhuma class
classe Carro pois ela é final

}
```

Posteriormente falaremos sobre herança, superclasse, subclasse e sobrescrever métodos.

Quando utilizamos o modificador final nos métodos, estamos dizendo que aquele método não pode ser sobrescrevido pelas subclasses. É usado quando queremos prevenir que algum comportamento de nossa classe não será sobrescrevido nas subclasses.

```
public class Carro {

	public final void acelerar(){// Método com o modificador final
	}
}

public class Sedan extends Carro{ //Classe Sedan herdando comportamentos da Carro

	public void acelerar(){//Erro de compilação, esse método é final na classe
pai, portanto, não pode ser sobrescrevido
	}
}
```

Quando utilizamos o modificador *final* nas variáveis, estamos dizendo que ela pode ser inicializada apenas uma vez, seja no ato da definição da variável ou em algum inicializador, tal como, um método construtor. Esse modificador também pode ser usado em variáveis locais e em variáveis de argumentos.

```
public class Carro {

	private final String cor; //Variável final cor

	private final String marca = "Volkswagen";//Variável final marca sendo
inicializada no ato de sua criação

	public Carro(String cor) {//Método construtor inicializado a variável final
cor
		this.cor = cor;
	}

	public void acelerar(final double velocidade){ {//Argumento final velocidade

		final double distancia = 2.0; //Variável local distancia final
	}

}
```

Depois de inicializada uma variável final o valor dela não pode ser alterado, essas variáveis também são conhecidas como constantes.

E referências de objetos com o modificador *final*? A mesma regra, veja o exemplo:

```
public class Carro {

	private String cor;
	//Métodos get/set
}

final Carro carro10 = new Carro();
```

Instanciamos a classe “Carro” em uma referência de objeto com o modificador final. Então não podemos alterar a instância que está apontada para a referência “carro10”. Se escrevermos o código abaixo, será apresentado um erro de compilação, dizendo que não podemos atribuir outro valor para a variável “carro10”.

```
carro10 = new Carro();//Erro de compilação
```

Porém, podemos alterar as características do objeto “carro10”, o código abaixo compila sem problemas. Nesse casso ocorre que não alteramos o objeto que estava referenciado pelo “carro 10”, e sim alteramos o estado do objeto que está referenciado.

```
carro10.setCor("Vermelho");
```

3.2.3 Modificador *abstract*

Esse modificador é utilizado apenas em classes e métodos. Em classes é usado para definir uma classe abstrata (Estudaremos posteriormente).

```
public abstract class Carro {

}
```

Em métodos, utilizamos esse modificador para criar métodos abstratos, esses métodos não possuem corpo e são utilizados em Classes Abstratas e Interfaces (Estudaremos mais sobre eles). Métodos sem corpo, não possuem as chaves “{ }”, ele é finalizado por ponto e virgula “;”.

```
public abstract void acelerar();
```

4 ASSOCIAÇÕES E POLIMORFISMO

4.1 COMPOSIÇÃO

Em Java, há duas maneiras para reusar classes existentes, denominadas, composição e herança. Com a composição, você define uma nova classe, que é composta de uma classe existente. Com a herança, você define uma nova classe baseada em uma classe existente, com modificações ou extensões. Como um exemplo de reuso via composição, suponha que nós temos uma classe denominada “Motor”, definida pelo código:

```
package br.aula.oo;
public class Motor {
```

```
	private double potencia;

	public double getPotencia() {
		return potencia;
	}

	public void setPotencia(double potencia) {
		this.potencia = potencia;
	}

}
```

Suponha que agora precisamos de uma classe denominada “Carro”, nós podemos reusar a classe “Motor” via composição. Nós dizemos que “Um carro é composto por motor” ou “Um Carro tem um motor”. A composição é um relacionamento referido com o termo “tem um”. A classe abaixo apresenta a classe “Carro” composta por um “Motor”.

```
package br.aula.oo;

public class Carro {

	private Motor motor; //Variável de referência do tipo Motor (Compõe a classe
Carro)
	private String cor;

	public Motor getMotor() {
		return motor;
	}
	public void setMotor(Motor motor) {
		this.motor = motor;
	}
	public String getCor() {
		return cor;
	}
	public void setCor(String cor) {
		this.cor = cor;
	}

}
```

Lembrando, a classe “Motor” ainda pode ser reusada em várias outras classes, como por exemplo: Um Avião tem um motor, um Barco tem um motor, dentre outros.

E como ficaria a modelagem de “Carro” composto por quatro ou mais rodas (estepe)?

```
package br.aula.oo;

public class Roda {

	private int aro;

	//Métodos get/set
}
```

Em Java, para dizer que uma classe é composta por mais de um item, podemos usar um "*Array*" ou um "*Collection*". *Array* e *Collection* são estruturas que permite armazenar uma coleção ou sequência de itens. A diferença entre eles é na inicialização, para inicializar um Array devemos primeiramente informar a quantidade de itens que será armazenado na estrutura. Já uma *Collection*, podemos inicializá-la sem informar a quantidade de itens, ou seja, podemos adicionar itens dinamicamente. O código abaixo apresenta a classe "Carro" composta por "Rodas".

A classe *java.util.Collection* é a interface raiz da hierarquia de coleções em Java. Geralmente trabalhamos com interfaces mais especificas que herdam as características da *Collection*, tais como:

java.util.List: Uma coleção ordenada que permite elementos repetidos;
java.util.Set: Uma coleção que não permite elementos duplicados;
java.util.Map: Uma coleção que trabalha o conceito chave/valor.

```
package br.aula.oo;

import java.util.List;

public class Carro {

	private Motor motor;
	private String cor;
	private List<Roda> rodas;//Uma lista do tipo Roda para armazenar N rodas

	//Métodos get/set

}
```

Ao definir "<Roda>" estamos dizendo que todos os elementos da coleção é do tipo roda (classe "Roda").

4.2 HERANÇA

Na POO, nós frequentemente organizamos nossas classes em hierarquia para evitar duplicação e reduzir a redundância. As classes de hierarquia inferior herdam todas as variáveis e métodos de hierarquias superiores. A classe na hierarquia inferior é chamada de subclasse. A classe na hierarquia superior é chamada de superclasse. Por exemplo, a Figura 2 apresenta uma subclasse denominada "Sedan", onde ela herda todas as características e comportamentos da classe "Carro". Os atributos "Cor" e "Peso" e os comportamentos "Acelerar" e "Frenar" são implementados na classe "Carro", e a classe "Sedan" pode reusa-los.

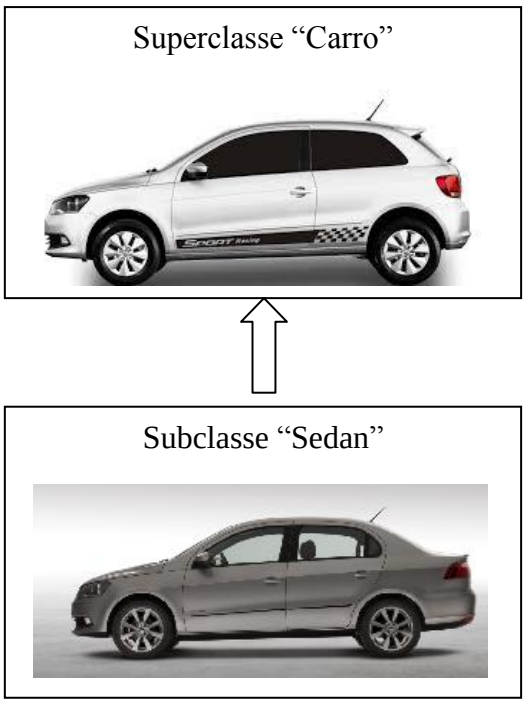

Figura 2 - Exemplo de Herança

A subclasse herda todos os métodos e variáveis de sua superclasse. Então a subclasse estende a superclasse para fornecer mais varáveis e métodos. Em Java definimos uma subclasse usando a palavra chave `extends`.

```
public class Carro { // Superclasse Carro
}

public class Sedan extends Carro {// Subclasse Sedan herda da classe Carro
}
```

Em Java uma classe pode estender apenas uma classe, ou seja, não é possível realizar uma herança múltipla. Herança múltipla implicaria em uma classe herdar métodos ou variáveis com um mesmo nome e de classes diferentes.

```
public class Sedan extends Carro, Automovel  {// Não compila
}
```

O código abaixo apresenta a classe “Carro” com atributos e métodos.

```
public class Carro {

	private Motor motor;
	protected String cor; //Modificador protected é visível apenas para as
subclasses
	private List<Roda> rodas;

	//Métodos get/set

	public void acelerar(){

	}

	public void frenar(){

	}

}
```

O código abaixo representa a classe “Sedan”. Essa classe herda todas características da classe “Carro” e ainda especifica uma característica “airbag” e um comportamento “acionarAirbag”

```
public class Sedan extends Carro {

	private String airbag;

	public void acionarAirbag(){

	}
}
```

O código abaixo apresenta uma instância da classe “Sedan”, invocando métodos da classe “Carro”, pois ela é uma subclasse.

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

	Sedan sedan = new Sedan();

	sedan.setCor("Vermelho");

	sedan.acelerar();

	sedan.acionarAirbag();
}
```

4.2.1 A palavra chave *super*

Lembre-se que dentro de uma definição de classe, podemos usar a palavra chave `this` para referenciar a "essa" instância. A palavra chave `super` é usada para referenciar a superclasse. A palavra chave `super` permite acessar a superclasse através da subclasse.

Podemos acessar uma superclasse para:

- Invocar algum método ou recuperar alguma variável. Exemplo:

```
public void acionarAirbag(){// Método da subclasse Sedan
      super.frenar(); // Acessando o método frenar implementado na
superclasse Carro
 }
```

- Inicializar o construtor da superclasse. Caso. Exemplo:

```
public Sedan(){
      super();// Inicializando o construtor da superclasse
 }
```

4.2.2 Mais sobre construtores

As subclasses não herdam os construtores da superclasse. Cada classe Java define o seu próprio construtor. No corpo de um construtor pode ser utilizado a palavra chave `super(<argumentos>);` para invocar o construtor de sua superclasse.

Atenção, se você usar o `super(<argumentos>);` ele deve ser a primeira declaração no construtor da subclasse. O código abaixo não compila, pois, a primeira declaração é "`int a = 2;`".

```
public Sedan(){
      int a = 2;
      super();// Inicializando o construtor da superclasse
}
```

Se você não inserir explicitamente a chamada do construtor `super();`o compilador Java invoca a declaração `super();` sem argumentos de sua superclasse, isso, se sua superclasse possuir um construtor sem argumentos.

```
public Sedan(){
      // Chamada explicita do construtor da superclasse super();
      int a = 2;
}
```

Se o construtor da superclasse possuir argumentos, você deve invoca-lo nos construtores das subclasses `super(<argumentos>);`.

4.2.3 A classe *java.lang.Object*

Todas as classes Java são derivadas (subclasses) de uma classe raiz comum chamada *java.lang.Object*. Essa classe define e implementa comportamentos comuns que são necessários para todos os objetos Java que rodam debaixo da JRE. Esses comportamentos comuns permite a implementação de características assim como *multi-threading* e *garbage collector*

> O *Garbage Collector* (coletor de lixo) é uma forma automática de gerenciamento de memória. Ele realiza uma limpeza dos objetos que estão referenciados na memória, porém, não estão em uso.

4.3 SOBREPOSIÇÃO E SOBRECARGA DE MÉTODOS

A subclasse herda todos os métodos e variáveis de sua superclasse. Ela pode usar os métodos e variáveis herdadas como elas são, ou pode também sobrescrever um método herdado para prover sua própria versão e esconder uma variável herdada definindo uma variável como o mesmo nome. Por exemplo, suponha-se que o porta-malas de um carro seja calculado pelo seu diâmetro dividido por cinco (“diâmetro/5”), conforme apresenta a classe abaixo.

```
public class Carro {

	protected double diamentro;

	public double calcularPortaMalas(){
		return this.diamentro / 5;
	}

}
```

Agora, precisamos de uma classe “Sedan” que herda os membros da classe “Carro”, porém, o cálculo do tamanho do porta-malas é diferente da classe “Carro”. Sendo assim, podemos sobrescrever o método “*calcularPortaMalas*”. Como apresenta o código:

```
public class Sedan extends Carro {

	@Override//Anotação opcional
	public double calcularPortaMalas(){//Método sobrescrito
		return super.diamentro / 4;
```

```
    }
}
```

Lembre-se para sobrescrever um método você precisa usar o mecanismo de herança. Para sobrescrever um método, ele precisa ter o mesmo nome e a mesma lista de argumentos. O código abaixo apresenta a execução de um método sobrescrito. Se o método "calcularPortaMalas" é invocado de um instância de "Carro" ("new Carro()"), o método a ser chamado é o declarado na subclasse "Carro", se o método for de uma instância de "Sedan" ("new Sedan()"), o método a ser chamado será a implementação sobrescrita na superclasse.

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

    Carro carro = new Carro();
    carro.setDiamentro(4.3);

    System.out.println(carro.calcularPortaMalas());

    Sedan sedan = new Sedan();
    sedan.setDiamentro(4.7);

    System.out.println(sedan.calcularPortaMalas());
}
```

Para sobrescrever métodos você deve ficar atento à algumas regras:

- Não é possível sobrescrever métodos com o modificador "*final";*
- Sé o método da superclasse for "*static*", então o método sobrescrito na subclasse deve também ser "*static*";
- Se o método da superclasse for "*public*", então o método da subclasse também deve ser "*public*";
- Se o método da superclasse for "*protected*", então o método da subclasse também deve ser "*protected*" ou "*public*";
- Se o método da superclasse tiver o modificador de acesso "*default*", então o método da subclasse também deve ser "*protected*", "*public*" ou "*default*".

4.3.1 A anotação *@Override*

O *@Override* é conhecida como uma anotação, que pede para o compilador verificar se existe tal método na superclasse para ser sobrescrevido. Ele ajuda você dizendo se

o método sobrescrevido existe com o mesmo nome e a mesma lista de argumentos na superclasse. Essa anotação é opcional, mas é sempre bom tê-la em seu código.

4.4 SOBRECARGA

A sobrecarga de métodos ocorre quando criamos dois ou mais métodos com o mesmo nome, porém, com a lista de argumentos diferentes. O código abaixo apresenta uma sobrecarga de métodos e construtores.

Não é possível ter métodos com a mesma assinatura em uma classe Java. Exceto quando é realizada uma sobreposição.

```
public class Carro {

	private double tempo;

	public Carro() {//Sobrecarga de construtor

	}

	public Carro(double tempo) {//Sobrecarga de construtor
		this.tempo = tempo;
	}

	public double calcularMedia(double km,double tempo){ {//Sobrecarga de método
		return km/tempo;
	}

	public double calcularMedia(double km){ {//Sobrecarga de método
		return km/this.tempo;
	}

}
```

Podemos criar o objeto usando dois construtores “new Carro()” e “new Carro(double)”. O método “calcularMedia” a ser invocado depende da lista de argumentos que será passado. Podemos chamar o método invocando “calcularMedia(double)” ou “calcularMedia(double,double)”.

4.5 CLASSE ABSTRATA

Um método abstrato é um método que possui apenas a assinatura (nome do método, lista de argumentos e tipo do retorno) sem implementação (sem corpo). Você pode usar a palavra chave `abstract` para declarar um método abstrato. Um método abstrato apenas

pode set usado em classes abstratas e interfaces.Uma classe abstrata deve ser declarada com o modificador **abstract**.

```
public abstract class Carro {//Classe abstrata

}
```

Em uma classe abstrata podemos ter zero ou vários métodos abstratos, não é obrigatório a definição de um método abstrato. O código abaixo apresenta uma classe abstrata.

```
public abstract class Carro {

	public double tempo;

	public double calcularMedia(double km){
		return km/tempo;
	}

	public abstract void acelerar();//Método abstrato
}
```

Uma classe abstrata não pode ser instanciada, em outras palavras não podemos criar objetos a partir de classes abstratas. O código "**new** Carro( )" lança um erro de compilação. Mas qual o sentido de criar uma classe se não podemos instanciá-la? Imagine que necessitamos de uma classe modelo denominada "Carro" que contém todas as características comuns a todos os carros, porém, o comportamento "acelerar" pode variar de carro para carro, então se define um método abstrato onde as subclasses irão implementar esse método de acordo com a sua necessidade. Agora a classe "Sedan" pode implementar o método acelerar de acordo com sua lógica.

Prosseguindo, para usar uma classe abstrata, você deve derivar uma subclasse de sua classe abstrata, ou seja, devemos criar uma classe que herde da classe "Carro". Ao estender uma classe abstrata automaticamente você deve implementar (sobrescrever) os métodos abstratos

```
public class Sedan extends Carro {

	@Override
	public void acelerar() {// Método implementado
		// Seu código aqui
	}

}
```

Como a classe "Sedan" é uma classe concreta, agora podemos instanciá-la ("**new** Sedan()").

Resumindo, a classe abstrata permite criar um *template* para o desenvolvimento, criando uma interface (contrato) comum para todas as subclasses, além de contribuir com o polimorfismo (será discutido posteriormente).

Algumas regras para o uso de classes abstratas:

- Uma classe abstrata não pode possui o modificador "*final*", bem como, um método abstrato também não pode ser definido como "*final*";
- Um método abstrato não pode ser estático "*static*";
- Um método abstrato não pode possuir o modificador de acesso "*private*", pois o método privado não é visível para a subclasse;
- Se uma subclasse "B" é abstrata e herda de outra classe abstrata "A", ela não é obrigada a implementar os métodos abstratos da classe "A". Caso uma subclasse "C" concreta herde de "B", então ela deve implementar os métodos estáticos da subclasse "B" e "A" caso exista.

4.6 INTERFACE

Realizando uma analogia, uma interface Java é uma superclasse 100% abstrata que define um conjunto de métodos abstratos que suas subclasses devem implementar. Uma interface possui apenas métodos "**`public abstract`**" (métodos com assinatura e sem implementação) e possivelmente constantes "**`public static final`**". Para definir uma interface devemos usar a palavra chave "**`interface`**" em vez de "**`class`**".

```
public interface Automovel {

	void acelerar();// Método público abstrato

	public abstract void frenar();// Método público abstrato
}
```

Conforme dito, todo método de uma interface é público e abstrato, mesmo se você não definir explicitamente.

Similar a uma classe abstrata, a interface não pode ser instanciada, porque ela é incompleta, seus métodos não possuem corpo (abstratos). Para usar a interface, devemos criar uma classe e usar a palavra chave "**`implements`**" para implementar a interface e definir uma lógica os métodos.

```
public class Carro implements Automovel{

	@Override
	public void acelerar() {
```

```
            // Seu código
        }

        @Override
        public void frenar() {
            // Seu código
        }

}
```

Outras classes que possuem comportamentos de “Automóvel”, como, caminhão, carreta, podem implementar a interface e fornece suas próprias implementações para os métodos abstratos.

Algumas regras para o uso de interface:

- Se uma classe abstrata “B” implementa uma interface “A”, ela não é obrigada a definir uma implementação para os métodos abstratos da interface. Caso uma classe concreta “C” estende a classe abstrata “B”, ela deve implementar os métodos abstratos da interface “A” e da classe “B”;
- Uma interface pode apenas estender interfaces, nesse caso pode haver herança múltipla entre interfaces;
- Uma classe pode implementar várias interfaces;
- Todo método implementado pela classe deve ser “`public`”.

O uso de interfaces contribui com o encapsulamento e o polimorfismo, permitindo que os componentes do seu projeto sejam menos acoplados.

4.7 POLIMORFISMO

A palavra polimorfismo significa várias formas. Por exemplo, na química, o carbono exibe o polimorfismo porque ele pode ser encontrado em mais de uma forma, como, grafite e diamante. Cada uma dessas formas possui suas próprias propriedades. Suponha, que o seu projeto tenha vários automóveis, assim como, carros, carros sedans, caminhões. Sendo assim, criaremos uma interface denominada “Automóvel”, que possui comportamentos comuns a todos automóveis.

```
public interface Automovel {

    void acelerar();
```

```
	public abstract void frenar();
}
```

Agora criaremos a classe “Carro” é uma automóvel, e irá implementar a interface “Automóvel”.

```
public class Carro implements Automovel{

	@Override
	public void acelerar() {
		System.out.println("Acelerar Carro");
	}

	@Override
	public void frenar() {
		System.out.println("Frenar Carro");
	}

}
```

Por último criaremos a classe “Sedan” que é um carro, e sendo um carro ele também é um automóvel. A classe “Sedan” estenderá a classe “Carro”, porém, a aceleração do carro sedan é diferente, então deveremos sobrescrever o método “acelerar”.

```
public class Sedan extends Carro {

	@Override
	public void acelerar() {
		System.out.println("Acelerar Sedan");
	}

	public void acionarABS() {
		System.out.println("ABS");
	}

}
```

Agora que o nossas classes estão criadas, vamos executá-las. Iremos criar referências de Automóveis e atribuir instâncias de subclasses, como Carro e Sedan.

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

	Automovel carro = new Carro();// Instância de carro
	carro.acelerar();// Saída - Acelerar Carro
	carro.frenar();// Saída - Frenar Carro

	Automovel sedan = new Sedan();// Instância de sedan
	sedan.acelerar();// Saída - Acelerar Sedan
	sedan.frenar();// Saída - Frenar Carro
}
```

A instância armazenada na referência, que irá ditar qual a chamada de método que será realizada. Mesmo se a referência for do tipo “Automóvel”, será a instância que definirá qual método será executado.

Como podemos atribuir instâncias de uma classe em uma referência totalmente diferente? Isso é o polimorfismo. O “Automóvel” pode ter várias formas, tais como, carro e sedan. Podemos criar uma referência de superclasse e atribuir uma instância de subclasse. Pois um “Carro” é um “Automóvel”, tal como, um “Sedan” é um “Carro” que por sequência é um “Automóvel”, então um sedan também é um automóvel.

Algumas orientações ao polimorfismo:

- Não é possui atribuir uma instância de superclasse em uma referência de subclasse. Ex: “`Carro carro = new Automovel();`”. Não compila pois um automóvel não é um carro, ele pode ser por exemplo um caminhão;
- Apenas é possível invocar membros que estão presentes na variável de instância. Por exemplo, “`Automovel sedan = new Sedan();`”. Não é possível invocar o método “Acionar Air Bag”, pois na interface automóvel ele não está presente.

4.8 PROGRAME PARA INTERFACES

Programe para interfaces e não para implementações. Muitos autores de padrões de projetos defendem a programação voltada para as interfaces, mas por quê?

Para entender, vamos primeiramente apreender os termos coesão e acoplamento. Em um projeto orientado a objetos, devemos ao máximo ter classes mais coesas e menos acopladas, pois, facilita a manutenção e possibilita o reuso adequado.

A coesão está relacionada com as operações de uma classe, uma classe com alta coesão realiza operações que são apenas de responsabilidade dela. Exemplo: Uma classe carro realiza a operação de frenar, acelerar, porém a ação de lavar não é de sua responsabilidade e deve ser criada uma nova classe para realizar essa ação.

O acoplamento está relacionado com o grau de dependência que uma classe possui com outras classes, quando menos dependência possui entre as classes, menos acoplado é o seu projeto. Por exemplo: Uma classe “A” acessa os atributos diretamente de uma classe “B”, se fosse necessário uma alteração na validação desses atributos na classe “B”, essa alteração impactaria na classe “A” pois estão fortemente acopladas.

O encapsulamento permite que você oculte os dados e métodos dentro classe, assim, procure sempre definir atributos como privados e métodos de acesso aos membros privados. Com o polimorfismo, podemos também esconder a lógica de nossa classe, pois, disponibilizando interfaces para o uso, os desenvolvedores não terão acesso direto a classe que está implementando a lógica. E qualquer manutenção na lógica não terá ou terá pouco impacto nas classes, pois, lembre-se estamos alterando a "Classe" e foi disponibilizada uma "Interface" para uso. Podemos ainda alterar completamente a lógica de um software, sem grandes alterações, para isso basta criamos uma nova classe que implemente a interface. Pois, lembre a interface pode ter várias formas (polimorfismo).

5 DICAS

5.1 ARGUMENTOS DE TAMANHO VÁRIAVEL

Em Java podemos ter um número de argumentos variável durante a chamada de um método.

Para isso, deveremos usar três pontos "...", veja a criação de um método.

```
	public static void imprime(int... valores){ // int... definição de uma argumento variável de inteiros

	}
```

Podemos definir qualquer tipo de dados com variável, porém, em um método podemos ter apenas um argumento de tamanho variável. Se precisar ter mais argumentos, o argumento de tamanho variável deve ser o último.

```
	public static void imprime(String nome,int... valores){

	}
```

Dentro do método o argumento de tamanho variável é tratado como um *array,*

```
public static void imprime(String nome,int... valores){

	if(valores != null){
```

```
            for (int i= 0;i <valores.length; i++) {
                  System.out.println(valores[i]);

            }
      }
}
```

Conforme dito, o tamanho do argumento é variável, então em sua chamada podemos passar zero ou vários números inteiros.

```
imprime("Eduardo");
imprime("Eduardo",1);
imprime("Eduardo",1,10,20);
imprime("Eduardo",1,20,10,40,20);
```

Se você não passar nenhum valor, então o *array* inicializará com `null`.

REFERÊNCIAS

FURGERI, Sergio. **Java 6 Ensino Didático**: Desenvolvendo e implementando aplicações. São Paulo: Érica, 2008.

HOCK-CHUAN, Chua. **Programming notes**. Disponível em: <http://www.ntu.edu.sg/home/ehchua/programming/index.html>. Acesso em: 04 jan. 2014.

SIERRA, Katherine; BATES, Bert. **SCJP Sun Certified Programmer**. Frederick: Mcgraw-hill, 2008.

ORACLE. **The Java Tutorials**. Disponível em: < http://docs.oracle.com/javase/tutorial/>. Acesso em: 04 jan. 2014.