Técnicas de Programação II

Aula 06 – Orientação a Objetos e Classes

Edirlei Soares de Lima
<edirlei.lima@uniriotec.br>

Orientação a Objetos

- O ser humano se relaciona com o mundo através do conceito de **objetos**.
- Damos **nomes** a esses objetos e os classificamos em **grupos** (**classes**).
- Os objetos possuem:
  - **Características** pelas quais os identificamos (**Atributos**);
    - O objeto **Pessoa** possui RG, nome, data de nascimento...
    - O objeto **Carro** possui tipo, cor, quantidade de portas...
  - **Finalidades** para as quais os utilizamos (**Comportamentos**);
    - Um objeto do tipo **Pessoa** pode andar, correr ou dirigir carros.
    - Um objeto do tipo **Carro** pode ligar, desligar, acelerar, frear.

Classes e Objetos

- A unidade fundamental da programação orientada a objetos é a **classe**.

- A classe define o que os objetos devem ter (exemplo: tipo, cor, placa e número de portas...) e quais operações eles podem realizar.

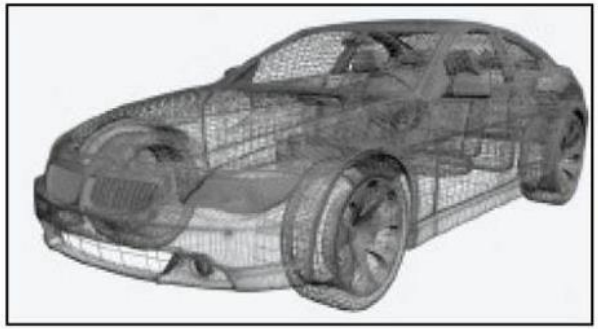
- As classes definem a **estrutura básica para a construção de objetos**.

Classe: Carro
Atributos: Tipo Cor Placa N° Portas ...
Métodos: Acelerar() Frear() TrocarMarcha(x) Buzinar() ...

Classes e Objetos

- Objetos são **instâncias** da **classe**.

Classe Carro



Objeto Carro1

Objeto Carro2

Objeto Carro3

Classes e Objetos

- A classe é o **modelo** ou **molde** de construção de objetos. Ela define as características e comportamentos que os objetos irão possuir.

- Sob o ponto de vista da programação orientada a objetos, um objeto não é muito diferente de uma **variável normal**.

- Um programa orientado a objetos é composto por um **conjunto de objetos** que interagem entre si.

Classes e Objetos

- Para manipularmos objetos em Java precisamos declarar **variáveis de objetos.**
- Uma variável de objeto é uma **referência** para um objeto.
- A **declaração de uma variável de objeto** é semelhante a declaração de uma variável normal:

```
Carro carro1;     /* carro1 não referencia nenhum objeto,
                     o seu valor inicial é null */
```

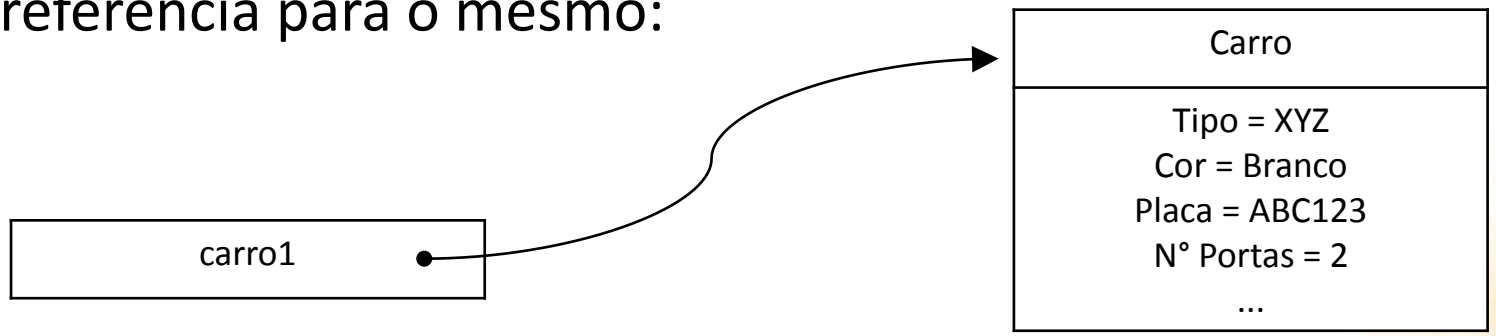
- A simples declaração de uma variável de objeto não é suficiente para a **criação de um objeto**.

Classes e Objetos

- A criação de um objeto deve ser explicitamente feita através do operador **new**:

```
Carro carro1;     /* carro1 não referencia nenhum objeto,
                     o seu valor inicial é null */

carro1 = new Carro();  /* instancia o objeto carro1 */
```

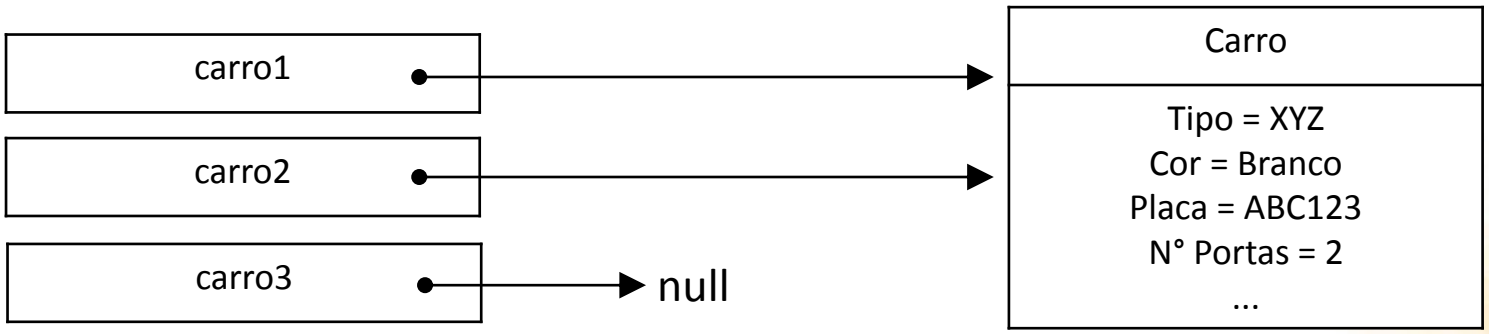
- O operador new **aloca o objeto em memória** e retorna uma referência para o mesmo:



Classes e Objetos

- Como as variáveis de objeto são referências, as operações de atribuição entre elas **não criam novos objetos**:

```
Carro carro1, carro2, carro3;

carro1 = new Carro();

carro2 = carro1;  /* carro2 passa a referenciar o mesmo
                     objeto referenciado por carro1 */
```



Classes e Objetos

- Um objeto expõe o seu **comportamento** através de métodos (funções).

- É possivel **invocar métodos** dos objetos instanciados:

```
Carro carro1 = new Carro();
Carro carro2 = new Carro();

carro1.mudarMarcha(2);
carro1.aumentaVelocidade(5);

carro2.mudarMarcha(4);
carro2.aumentaVelocidade(10);
```

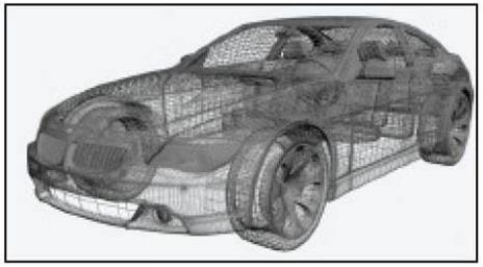
Criação de Classes

```
public class Carro {

    private String tipo;
    private String cor;
    private String placa;
    private int portas;
    private int marcha;
    private double velocidade;

    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Frear()
    {
        velocidade -= marcha * 10;
    }

    ...
}
```

Declaração

Atributos

Métodos



Classe: Carro
Atributos: Tipo Cor Placa N° Portas ...
Métodos: Acelerar() Frear() TrocarMarcha(x) Buzinar() ...

Modificadores de Acesso

```
public class Carro {

    public double velocidade;

    private String cor;

    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Frear()
    {
        velocidade -= marcha * 10;
    }
}
```

O modificador **public** declara que a classe pode ser usada por qualquer outra classe. Sem public, uma classe pode ser usada somente por classes do mesmo pacote

O modificador **public** declara que o atributo pode ser acessado por métodos externos à classe na qual ele foi definido.

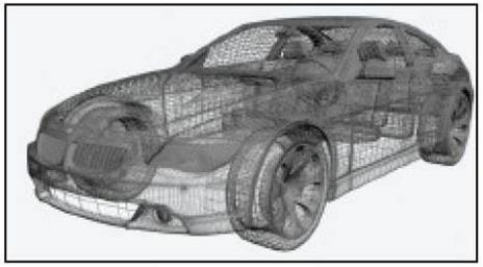
O modificador **private** declara que o atributo não pode ser acessado por métodos externos à classe na qual ele foi definido.

O modificador **public** declara que o método pode ser executado por métodos externos à classe na qual ele foi definido.

Encapsulamento!

Métodos Set/Get

```
public class Carro {
    ...

    public double GetVelocidade()
    {
        return velocidade;
    }

    public int GetMarcha()
    {
        return marcha;
    }

    public void TrocarMarcha(int novaMarcha)
    {
        marcha = novaMarcha;
    }

    ...
}
```

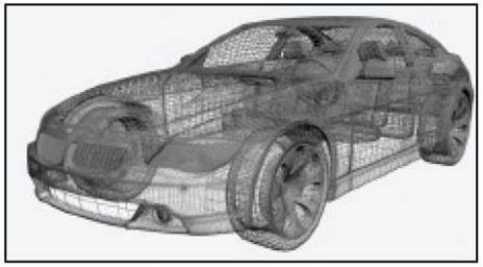


Classe: Carro
Atributos: Tipo Cor Placa N° Portas ...
Métodos: Acelerar() Frear() TrocarMarcha(x) Buzinar() ...

Método Construtor

Mesmo nome da classe e sem nenhum retorno.

```
public class Carro {

    ...

    public Carro(String ntipo, String ncor,
                 String nplaca, int nportas)
    {
        tipo = ntipo;
        cor = ncor;
        placa = nplaca;
        portas = nportas;

        marcha = 1;
        velocidade = 0;
    }

    ...
}
```



Classe: Carro
Atributos: Tipo Cor Placa N° Portas ...
Métodos: Acelerar() Frear() TrocarMarcha(x) Buzinar() ...

Métodos Estáticos

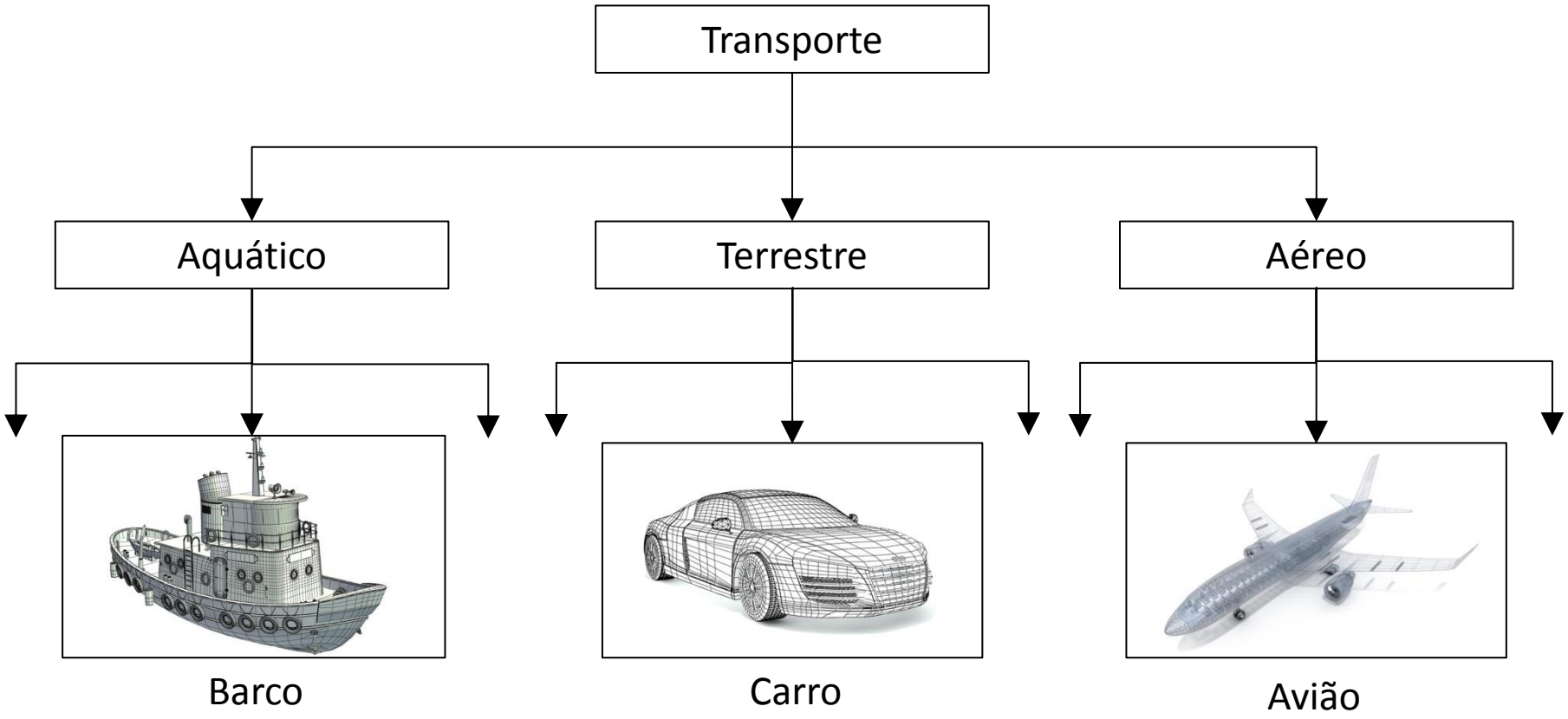
Método estático para calcular a distancia entre dois carros.

```
public class Carro {

  ...

  public static double Distancia(Carro c1,
                                 Carro c2)
  {
    return Math.sqrt(Math.pow(c1.GetX() - c2.GetX(), 2) +
                     Math.pow(c1.GetY() - c2.GetY(), 2));
  }

  ...
}
```

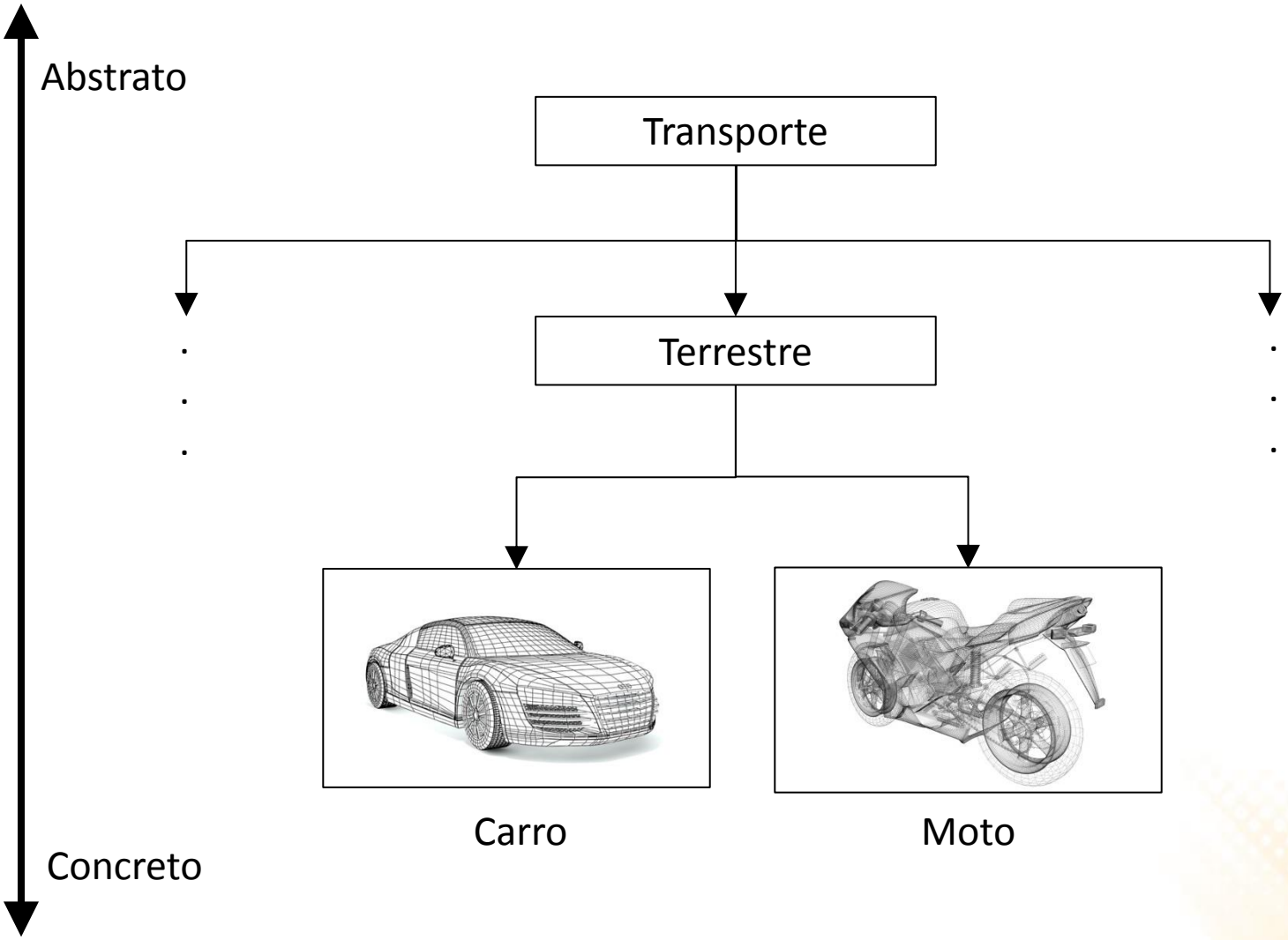
```
System.out.println("Distancia: " + Carro.Distancia(MeuCarro1,
                                                   MeuCarro2));
```

Herança

- Herança é um mecanismo que permite a uma classe herdar todos os atributos e métodos de outra classe.



Herança



Herança

```
public class Transporte {
    protected int capacidade;
    protected int velocidade;

    public Transporte()
    {
        velocidade = 0;
    }
    public int GetCapacidade()
    {
        return capacidade;
    }
    public void SetCapacidade(int cap)
    {
        capacidade = cap;
    }
    public int GetVelocidade()
    {
        return velocidade;
    }
}
```

O modificador **protected** determina que apenas a própria classe e as classes filhas poderão ter acesso ao atributo.

Classe: Transporte
Atributos: Capacidade Velocidade
Métodos: GetCapacidade() SetCapacidade(c) GetVelocidade()

Herança

```
public class Terrestre extends Transporte{

    protected int numRodas;

    public Terrestre(int rodas)
    {
        super();
        numRodas = rodas;
    }

    public int GetNumRodas()
    {
        return numRodas;
    }

    public void SetNumRodas(int num)
    {
        numRodas = num;
    }
}
```

A classe Terrestre **herda** as características da classe Transporte.

Chamada ao **método construtor** da classe pai (Transporte).

Classe: Terrestre
Atributos: N° Rodas
Métodos: GetNumRodas() SetNumRodas(n)

Herança

```
public class Carro extends Terrestre{
    private String cor;
    private String placa;
    private int marcha;

    public Carro(String ncor, String nplaca)
    {
        super(4);
        cor = ncor;
        placa = nplaca;
        marcha = 0;
    }
    public int GetMarcha()
    {
        return marcha;
    }
    public void TrocarMarcha(int novaMarcha)
    {
        marcha = novaMarcha;
    }
    ...
```

A classe Carro **herda** as características da classe Terrestre.

Chamada ao **método construtor** da classe pai (Terrestre).

Classe: Carro
Atributos: Cor Placa Marcha
Métodos: GetMarcha() TrocaMarcha(n) Acelerar() Frear()

Herança

```
    ...


    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Frear()
    {
        velocidade -= marcha * 10;
    }

}
```

Classe: Carro
Atributos: Cor Placa Marcha
Métodos: GetMarcha() TrocaMarcha(n) Acelerar() Frear()

Herança

```
public class Moto extends Terrestre{
    private String cor;
    private String placa;
    private int marcha;

    public Moto(String ncor, String nplaca)
    {
        super(2);
        cor = ncor;
        placa = nplaca;
        marcha = 0;
    }
    public int GetMarcha()
    {
        return marcha;
    }
    public void TrocarMarcha(int novaMarcha)
    {
        marcha = novaMarcha;
    }
    ...
```

A classe Carro **herda** as características da classe Terrestre.

Chamada ao **método construtor** da classe pai (Terrestre).

Classe: Moto
Atributos: Cor Placa Marcha
Métodos: GetMarcha() TrocaMarcha(n) Acelerar() Frear()

Herança

```
    ...


    public void Acelerar()
    {
        velocidade += marcha * 5;
    }

    public void Frear()
    {
        velocidade -= marcha * 5;
    }

}
```

Classe: Moto
Atributos: Cor Placa Marcha
Métodos: GetMarcha() TrocaMarcha(n) Acelerar() Frear()

Herança

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

  Carro meuCarro = new Carro("Vermelho", "ABC-1234");
  Moto minhaMoto = new Moto("Preto", "XYZ-9876");

  meuCarro.TrocarMarcha(1);
  meuCarro.Acelerar();
  meuCarro.Acelerar();

  minhaMoto.TrocarMarcha(minhaMoto.GetMarcha() + 1);
  minhaMoto.Acelerar();
  minhaMoto.TrocarMarcha(minhaMoto.GetMarcha() + 1);
  minhaMoto.Acelerar();

  System.out.println("Velocidade do Carro: "+meuCarro.GetVelocidade());
  System.out.println("Velocidade da Moto: "+minhaMoto.GetVelocidade());

}
```

Classe `Object`

- A classe **`java.lang.Object`** é a ancestral de qualquer classe em Java; isto é, toda classe definida em Java herda implicitamente as propriedades e métodos de `Object`.

- Um dos método herdados é o `equals()`. Porém, o método implementado em `Object` gera o mesmo resultado que o operador ==.

- Caso queiramos implementar o nosso próprio conceito de igualdade, devemos **sobrescrever** o método `equals()`.

Overriding

```
public class Carro extends Terrestre{

    ...

    @Override
    public boolean equals(Object ob)
    {
        if (ob instanceof Carro)
        {
            Carro c = (Carro)ob;
            if (c.placa.equals(placa) && c.cor.equals(cor))
                return true;
        }
        return false;
    }

    ...

}
```

Anotação para indicar ao compilador que o método equals da classe Object será sobrescrito.

O método recebe um objeto da classe Object que deve ser explicitamente convertido para um objeto Carro.

O comando **instanceof** testa se um determinado objeto é de uma classe especifica.

Polimorfismo

- Polimorfismo refere-se a capacidade de objetos assumirem várias formas.

- Em Java existem dois tipos de polimorfismo:

  - **Overloading (sobrecarga):** permite que uma classe possa ter vários métodos com o mesmo nome, mas com assinaturas distintas.

  - **Overriding (sobrescrita):** permite que uma classe possua um método com a mesma assinatura (nome, tipo e ordem dos parâmetros) que um método da sua superclasse (o método da classe derivada sobrescreve o método da superclasse).

Polimorfismo - Overloading

- Overloading de métodos:

```
public class Carro extends Terrestre{

    ...

    public void Acelerar()
    {
        velocidade += marcha * 10;
    }

    public void Acelerar(double forca)
    {
        velocidade += marcha * 10 * forca;
    }

    ...
}
```

Polimorfismo - Overloading

- Overloading de métodos construtores:

```
public class Carro extends Terrestre{

    ...

    public Carro()
    {
        super(4);
    }

    public Carro(String ncor, String nplaca)
    {
        super(4);
        cor = ncor;
        placa = nplaca;
        marcha = 0;
    }
    ...
}
```

Polimorfismo - Overriding

```
public class Terrestre extends Transporte{

    ...

    public void Acelerar()
    {
        velocidade += 1;
    }
}
```

```
public class Carro extends Terrestre{

    ...

    @Override
    public void Acelerar()
    {
        velocidade += marcha * 10;
    }
}
```

O método Acelerar da classe Carro sobrescreve o método Acelerar da classe pai Transporte.

Modificador `final`

- O modificador `final` pode ser utilizado em **atributos** para definir constantes:

```
public class Aviao{

    private final int codigo = 25;
    private final String nome = "Aviao";

    ...
}
```

- Ele também pode ser utilizado na **declaração da classe** para impedir a criação de subclasses:

```
public final class Aviao{

    ...
}
```

Modificador `final`

- O modificador `final` também pode ser utilizado em métodos para impedir que eles possam ser sobreescritos:

```
public class Aviao{

    ...

    public final void Acelerar()
    {
        velocidade += marcha * 10;
    }

    ...

}
```

Classes Abstratas e Operações Abstratas

- O modificador `abstract` pode ser utilizado na declaração de uma classe para definir que ela é abstrata.
  - Uma classe abstrata normalmente possui um ou mais métodos abstratos.

- Um método abstrato não possui implementação, seu propósito é obrigar as classes descendentes a fornecerem implementações concretas das operações.

- Uma classe abstrata não pode ser instanciada diretamente.

Classes Abstratas e Operações Abstratas

```
public abstract class Terrestre extends Transporte{

    ...

    public abstract void Acelerar();
}
```

```
public class Carro extends Terrestre{

    ...

    @Override
    public void Acelerar()
    {
        velocidade += marcha * 10;
    }
}
```

Ocorrerá um erro de compilação se a classe Carro não implementar o método Acelerar.

Interface

- Interfaces permitem expressar comportamentos de classes sem definir as suas implementação.
  - Os métodos somente serão efetivamente implementados pelas classes que implementarem a interface;
  - Uma interface não pode ser instanciada diretamente;

```
public interface Voador{
    void Voar(double tempo);
}
```

```
public class Aviao implements Voador{

    public void Voar(double tempo)
    {
        ...
    }
}
```

Ocorrerá um erro de compilação se a classe Aviao não implementar o método Voar.

Interface

```
public interface Voador{
    public void Voar(double tempo);
}
```

```
public class Aviao implements Voador{

    public void Voar(double tempo)
    {
        ...
    }
}
```

```
public class DiscoVoador implements Voador{

    public void Voar(double tempo)
    {
        ...
    }
}
```

Interface

- Uma classe pode implementar várias interfaces:

```
public interface Voador{
    public void Voar(double t);
}
```

```
public interface Animal{
    public void Comer(int qtd);
    public void Dormir(double t);
}
```

```
public class Ave implements Voador, Animal{

    ...

    public void Voar(double t){
        ...
    }
    public void Comer(int qtd){
        ...
    }
    public void Dormir(double t){
        ...
    }
}
```

Interface – Utilização

- A API Java implementa algumas das suas funcionalidades através de interfaces.

- Exemplo: método `sort` da classe `Arrays`.

```
public interface Comparable{

    public int compareTo(Object o);

}
```

Interface – Utilização

```
public class Empregado implements Comparable{

    private String nome;
    private double salario;

    ...

    public int compareTo(Object o)
    {
       Empregado e = (Empregado)o;
       if (this.salario > e.salario)
           return 1;
       else if (this.salario < e.salario)
           return -1;
       else
           return 0;
    }

    ...

}
```

Interface – Utilização

```
...

public static void main(String[] args)
{
    Empregado[] lista={new Empregado("Joao",50.0),
                       new Empregado("Ana",30.0),
                       new Empregado("Paula",100.0),
                       new Empregado("Carlos",10.0)};

    Arrays.sort(lista);

    for(Empregado e:lista)
       System.out.println(e.getNome()+" "+e.getSalario());

}

...
```

Exercícios

Lista de Exercícios 07 – Orientação a Objetos e Classes

http://uniriodb2.uniriotec.br