

<LAB365>

Herança

AGENDA

- Herança
- Hierarquia de classes
- Herança e Composição
- super
- Override
- Classe Object

Herança

- Herança é um dos principais recursos de reutilização de código em POO:
 - Uma classe pode herdar atributos e métodos de uma classe já existente.
 - Atributos e métodos de uma classe já existente encontram-se disponíveis na nova classe sem que seja necessário declará-los explicitamente.
- Aumenta a produtividade do desenvolvimento.
- Pode melhorar a manutenibilidade de um programa.

Herança

- A “classe já existente”, ou “parent class”, ou “classe mãe” é oficialmente chamada de “**superclasse**”.
- A “classe filha”, ou “classe herdeira” chama-se “**subclasse**”.
- Portanto, a subclasse **herda** os atributos e métodos da superclasse.

Herança

- Além dos atributos e métodos herdados, a subclasse usualmente adiciona seus próprios atributos e métodos.
- Subclasse é mais específica, uma **extensão** da superclasse.

Herança

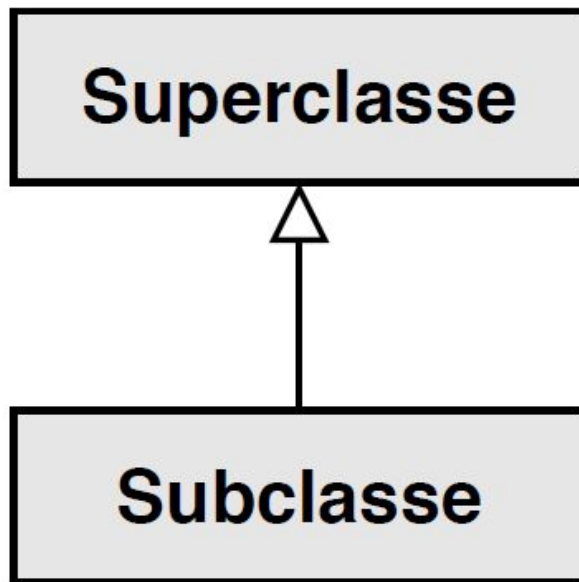
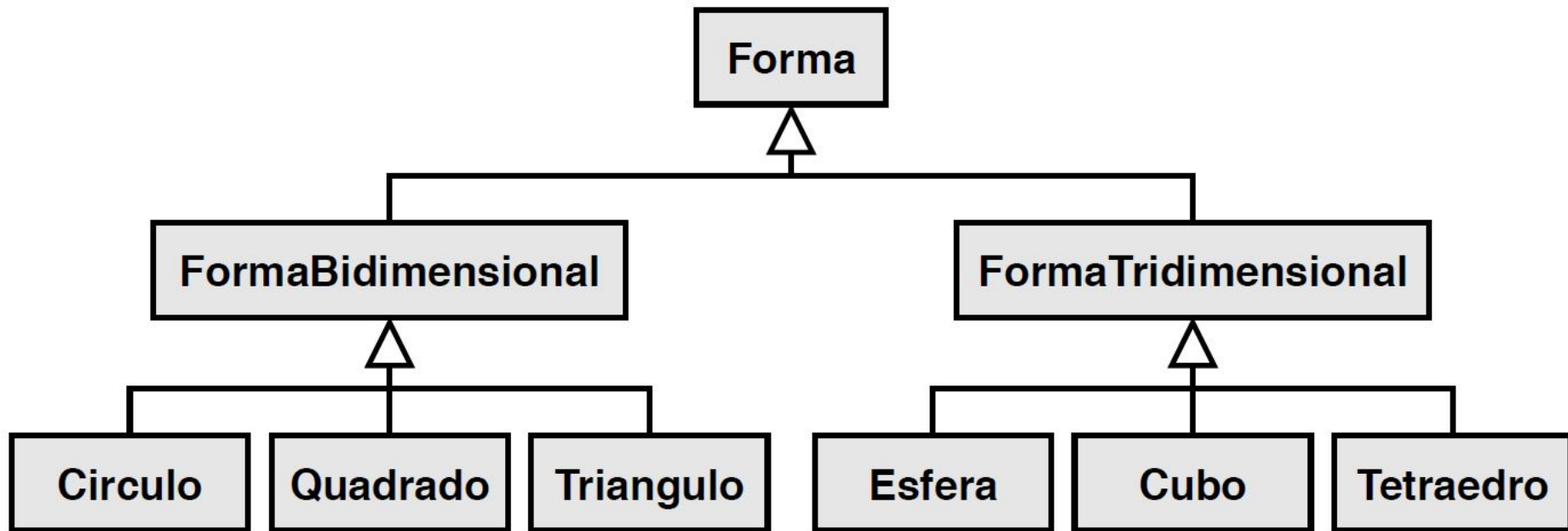


Diagrama de classes UML

Herança

- Uma superclasse pode ser subclasse de outra superclasse
- Subclasses podem se tornar superclasses para futuras subclasses
- Superclasse direta é a superclasse a partir da qual a subclasse herda explicitamente
- Superclasse indireta é qualquer superclasse acima da superclasse direta na hierarquia de classes

Herança



Herança

- Existem, basicamente, dois tipos de relações entre classes:
- **Herança**, que estabelece uma relação de especialização entre duas classes
 - Superclasse corresponde ao conceito mais genérico.
 - Subclasse corresponde ao conceito mais específico.
 - Também chamada de relação **é um**.
- Exemplo: Carro é uma subclasse de MeioDeTransporte. Ou seja: Carro **é um** MeioDeTransporte.

Herança

- **Composição**, que define um relacionamento entre duas classes no qual uma instância de uma classe agrupa uma ou mais instâncias da outra classe
 - Também chamada de relação **tem um**.
- Exemplo: Carro é composto de Motor. Ou seja, Carro **tem um** Motor.

Herança

- Para declarar que uma classe é subclasse de outra, utilizamos a palavra-chave “extends”:

```
public class ContaCorrente extends Conta {
```

```
public class ContaPoupanca extends Conta {
```

Herança

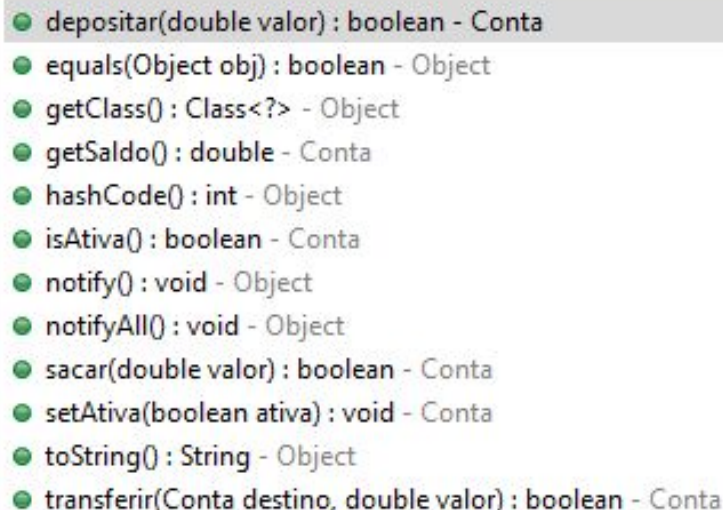
- Palavra-chave “extends” estabelece a relação de herança em Java.
 - Java não suporta herança múltipla, portanto apenas uma classe pode aparecer depois de “extends”.
- Pode ser interpretado de diferentes formas, todas com o mesmo sentido final:
 - Classe ContaCorrente estende ou especializa a classe Conta
 - Classe ContaCorrente herda os atributos e métodos da classe Conta
 - Classe ContaCorrente é um tipo de Conta
 - Classe ContaCorrente é uma subclasse direta de Conta
 - Classe Conta é uma superclasse direta de ContaCorrente

Herança

- As classes ContaCorrente e ContaPoupanca possuem os mesmos atributos e métodos que a classe Conta:

```
ContaCorrente cc = new ContaCorrente();
```

```
cc.
```



- depositar(double valor) : boolean - Conta
- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- getSaldo() : double - Conta
- hashCode() : int - Object
- isAtiva() : boolean - Conta
- notify() : void - Object
- notifyAll() : void - Object
- sacar(double valor) : boolean - Conta
- setAtiva(boolean ativa) : void - Conta
- toString() : String - Object
- transferir(Conta destino, double valor) : boolean - Conta

Herança

- Podemos criar novos atributos próprios da ContaCorrente:

```
public class ContaCorrente extends Conta {  
    private double limite;  
  
    public double getLimite() {  
        return limite;  
    }  
  
    public void setLimite(double limite) {  
        this.limite = limite;  
    }  
}
```

Herança

- Será que conseguimos acessar os atributos da superclasse?
- Se eu quiser, por exemplo, criar um método “getSaldoComLimite()” que retorna o valor do saldo + limite, vai funcionar?

Herança

- Relembrando:
- Atributos e métodos **private** de uma classe são acessíveis apenas nos métodos da própria classe.
- Atributos e métodos **public** de uma classe são acessíveis de qualquer método de qualquer classe do programa, a partir de uma referência a um objeto desta classe ou de uma de suas subclasses.
- Atributos e métodos com **nenhum modificador** especificado (padrão) são acessíveis apenas para classes do mesmo pacote.

Herança

- Modificador de acesso **protected** oferece um nível intermediário de acesso.
- Atributos e métodos **protected** de uma classe são acessíveis por métodos da própria classe, por métodos de suas subclasses e por métodos de outras classes no mesmo pacote.
- Portanto, uma subclasse pode acessar atributos e métodos **public** e **protected** diretamente pelos seus nomes, mesmo que de pacotes diferentes.

Herança

- Modificador de acesso **protected** oferece um nível intermediário de acesso.
- Atributos e métodos **protected** de uma classe são acessíveis por métodos da própria classe, por métodos de suas subclasses e por métodos de outras classes no mesmo pacote.
- Portanto, uma subclasse pode acessar atributos e métodos **public** e **protected** diretamente pelos seus nomes, mesmo que de pacotes diferentes.

Prática

- Vamos criar uma classe Vendedor, com os atributos: String nome, String sobrenome, double valorVendidoNoMes, double salarioBase.
- Inserimos métodos getters e setters na classe.
- Vamos deixar a classe sem construtor por enquanto.

Prática

- Agora vamos criar uma subclasse de Vendedor, chamada VendedorComissionado.
- Essa classe deve ter, além dos atributos de Vendedor, um novo chamado taxaComissao, de tipo double.
 - A classe deve ter get e set para esse novo atributo.
- A classe deve validar a taxa de comissão, que deve ser um valor entre 0.0 e 1.0.
- Deve haver um método “getSalario” para retornar o cálculo do salário do vendedor comissionado: $\text{salário base} + (\text{taxa comissão} * \text{valor vendido no mês})$.

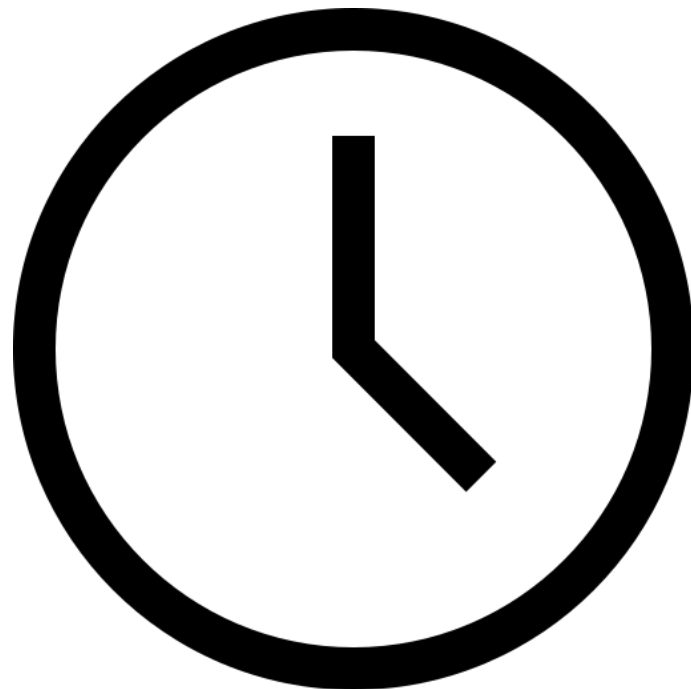
INTERVALO!

Finalizamos o nosso primeiro período de hoje. Que tal descansar um pouco?!

Nos vemos em 20 minutos.

Início: 20:25

Retorno: 20:45



Herança

- **Construtores** da superclasse **não são herdados** pela subclasse.
- Se a superclasse não definir um construtor, a subclasse pode ou não definir o seu próprio.
- Mas caso a superclasse defina um construtor com parâmetros, a subclasse obrigatoriamente precisa definir um construtor que execute o construtor de sua classe mãe, passando os devidos argumentos.

Herança

- Isso acontece porque ao inicializar um objeto de uma subclasse, o construtor da superclasse sempre é invocado.
- Lembrando: mesmo quando não definimos um construtor, existe um construtor vazio.
 - Ou seja: ao invocar o construtor de uma subclasse, este sempre invoca o construtor da superclasse. Mesmo que eles não estejam explícitos.

Herança

- Podemos ver isso na prática:

```
public Conta() {  
    System.out.println("Construtor da superclasse.");  
}
```

```
public class ContaCorrente extends Conta {  
  
    public ContaCorrente() {  
        System.out.println("Construtor da subclasse.");  
    }  
}
```


Herança

- Então se a superclasse não possuir um construtor explícito, ou se possuir um construtor que não recebe parâmetros, este é **implicitamente** chamado pelo construtor da subclasse.
- Dá para entender como isso é possível, já que o construtor não recebe nenhum parâmetro.
- Mas e se receber?

Herança

- Esse construtor continua sendo invocado implicitamente?

```
public class Conta {  
    private Cliente titular;  
    protected double saldo;  
    private boolean ativa;  
  
    public Conta(String nomeTitular,  
                  String sobrenomeTitular,  
                  String cpfTitular) {  
        this.titular = new Cliente(nomeTitular,  
                                     sobrenomeTitular, cpfTitular);  
    }  
}
```

Herança

- Vemos que não. E como resolver?

```
public class ContaCorrente extends Conta {  
    private double limite;  
  
    public ContaCorrente() {  
        System.out.println("Construtor da subclasse.");  
    }  
}
```

```
public ContaCorrente() {
```

```
Sys
```

```
}
```

✖ Implicit super constructor Conta() is undefined. Must explicitly invoke another constructor

Press 'F2' for focus

Herança

- Como diz a mensagem de erro, não existe mais um construtor “Conta()” para ser invocado implicitamente.
- Portanto, precisamos invocar **explicitamente** um construtor válido.

```
public ContaCorrente() {  
    Sys  
}
```

✖ Implicit super constructor Conta() is undefined. Must explicitly invoke another constructor

Press 'F2' for focus

Herança

- Para isso, utilizamos a palavra-chave “super”.

```
public ContaCorrente(String nome, String sobrenome, String cpf) {  
    super(nome, sobrenome, cpf);  
}
```

Herança

- A palavra-chave “super” referencia a superclasse direta.
- Pode ser utilizada para invocar o construtor, métodos ou atributos da superclasse.
 - Caso for usada em um construtor da subclasse, deve ser a primeira linha.
- Vimos que atributos privados da superclasse não são acessíveis pelas subclasses. Portanto para inicializar esses atributos, é necessário invocar o construtor da superclasse.

Herança

- Muitas vezes, a subclasse precisa de uma versão personalizada de um método herdado da superclasse.
- A subclasse pode **sobrescrever** o método da superclasse.
 - Utiliza a **mesma assinatura**, mas muda o corpo.
- Anotação **@Override** sinaliza para o compilador que um método está sendo sobrescrito.

```
@Override  
public double getSaldo() {  
    return this.limite + this.saldo;  
}
```

Herança

- Não é obrigatório utilizar a anotação **@Override** para sobrescrever um método, mas é mais seguro fazê-lo.
- Utilizando **@Override**, o compilador verifica se existe algum método com a mesma assinatura nas superclasses.
- Ou seja, caso o método sobrescrito sofra alguma alteração futura na sua assinatura em alguma superclasse, o compilador avisará.

Herança

- E se eu sobrescrever um método na minha subclasse, mas em algum momento precisar executar o método original, e não a versão da subclasse?
- É possível invocar um método da superclasse utilizando a palavra-chave “super”.

```
@Override  
public double getSaldo() {  
    return this.limite + super.getSaldo();  
}
```

Herança

- Outro exemplo: se tivermos um método “imprimeSaldo” na classe Conta.

```
public void imprimeSaldo() {  
    System.out.printf("O cliente %s %s possui R$ %.2f de saldo em sua conta.%n",  
        this.titular.nome, this.titular.sobrenome, this.saldo);  
}
```

- Suponha que precisamos criar uma nova versão de “imprimeSaldo” na classe ContaCorrente, exibindo a mesma frase do método original, porém acrescido da informação de limite.
- Não queremos repetir um código já escrito. Como fazer?

Herança

- Utilizamos o “super”:

```
@Override
public void imprimeSaldo() {
    super.imprimeSaldo();
    System.out.printf("O limite de sua Conta Corrente é de R$ %.2f.",
        this.limite);
}
```

- No método da subclasse, primeiro invocamos o método da superclasse.
- Logo em seguida, fazemos os acréscimos da subclasse.

Herança

- Todas as classes do Java herdam direta ou indiretamente da classe Object.
- Isto implica que os 11 métodos da classe Object são herdados por todas as classes.
- Alguns destes métodos são amplamente utilizados e, em geral, sobrescritos para fornecer um comportamento personalizado.
 - Principalmente equals e toString.

Herança

Método	Descrição
protected Object clone()	Cria e retorna uma cópia deste objeto
boolean equals(Object obj)	Indica se algum objeto é “igual” a este objeto
String toString()	Retorna uma representação textual deste objeto
Class<?> getClass()	Retorna a classe em tempo de execução deste objeto
int hashCode()	Retorna o código hash para o objeto
protected void finalize()	Executado quando o objeto é coletado para descarte (descontinuado)

Herança

Método	Descrição
<code>void notify()</code>	Acorda uma única thread aguardando no monitor deste objeto
<code>void notifyAll()</code>	Acorda todas as threads aguardando no monitor deste objeto
<code>void wait()</code>	Faz com que a thread atual aguarde até ser acordada
<code>void wait(long timeMs)</code>	Faz com que a thread atual aguarde até ser acordada ou que uma quantidade de tempo passe
<code>void wait(long timeMs, int nanos)</code>	Faz com que a thread atual aguarde até ser acordada ou que uma quantidade de tempo passe

Herança

- Sobrescrevendo toString() na classe Conta:

```
@Override
public String toString() {
    return String.format("%s: %s %s%n%s: R$ %.2f%n%s: %b",
        "Titular", this.titular.nome, this.titular.sobrenome,
        "Saldo", this.saldo,
        "Conta ativa", this.ativa);
}
```

- Então podemos só chamar um objeto Conta dentro de um print, por exemplo, para visualizar sua representação em String.

```
Conta conta = new Conta("João", "Oliveira", "1");

System.out.println(conta);
```

Herança

- Sobrescrevendo equals() na classe Endereco:

```
class Endereco {  
    String logradouro;  
    String numero;  
    String complemento;  
    String cep;  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Endereco that = (Endereco) o;  
        return this.logradouro.equals(that.logradouro) &&  
            this.numero.equals(that.numero) &&  
            this.complemento.equals(that.complemento) &&  
            this.cep.equals(that.cep);  
    }  
}
```


Material complementar

- Herança em Java:
<https://www.devmedia.com.br/entendendo-e-aplicando-heranca-em-java/24544>
- Herança de construtores e sobrescrita de métodos (Override):
<https://www.javaprogressivo.net/2012/10/heranca-de-construtores-e-override.html>
- Palavra-chave “super”:
<https://www.programiz.com/java-programming/super-keyword>
- Classe Object: <https://www.javatpoint.com/object-class>

<LAB365>

Polimorfismo

AGENDA

- Polimorfismo

Polimorfismo

- Vimos que uma variável de tipo Conta, por exemplo, guarda a referência para o objeto, nunca o objeto em si.
- Em herança, vimos que toda ContaCorrente é uma Conta, pois é uma extensão desta.
- Se um cliente quiser abrir uma Conta no nosso banco, ele pode abrir uma ContaCorrente, porque ContaCorrente é uma Conta.

Polimorfismo

- Portanto:

```
ContaCorrente cc = new ContaCorrente("João Victor",  
                                         "Mendes", "1", 50, 100);
```

```
Conta conta = cc;
```

```
conta.depositar(50);
```

Polimorfismo

- Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas.
- O que não quer dizer que ele muda o seu tipo.
- Pelo contrário, um objeto nasce de um tipo e morre daquele tipo.
- O que pode mudar é a **forma** como nos referimos a ele.

Polimorfismo

- Utilizando o exemplo anterior de atribuir uma variável de tipo ContaCorrente a outra variável de tipo Conta, o que acontece se invocarmos o método “getSaldo”, que foi sobrescrito na subclasse?
- Qual versão do método é executada?

Polimorfismo

```
ContaCorrente cc = new ContaCorrente("João Victor",  
    "Mendes", "1", 50, 100);
```

```
Conta conta = cc;
```

```
conta.depositar(50);
```

```
System.out.println(conta.getSaldo());
```

```
<terminated> MeuBanco (1) [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe
```

```
Valor de 50,00 depositado com sucesso!  
200.0
```


Polimorfismo

- No Java, a invocação de método sempre vai ser decidida em tempo de execução.
- O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado.
- Sempre relacionando com sua classe de verdade, e não com a que estamos usando para referenciá-lo.
- Apesar de estarmos nos referenciando a essa ContaCorrente como sendo uma Conta, o método executado é o da ContaCorrente.

Polimorfismo

- A ideia do polimorfismo é conseguirmos manipular objetos das subclasses como se fossem objetos da superclasse.
- Por isso, não podemos realizar a atribuição inversa.

```
cc = conta;
```

Type mismatch: cannot convert from Conta to ContaCorrente

- Nem atribuir uma variável de outro tipo que não tenha relação de hierarquia.

```
Endereco end = new Endereco("Rua xyz", "123",  
                             "Bloco apto", "88777000");
```

```
Conta conta = end;
```

Polimorfismo

- Por que criar uma variável ContaCorrente e referenciá-la como uma Conta?
- Na verdade, o que costuma acontecer é termos um método que recebe um argumento do tipo Conta, e nós podemos passar uma variável tipo ContaCorrente, ou de qualquer outra subclasse de Conta.
- Dessa forma, podemos escrever um método que será compartilhado por todas as subclasses.
- É essa lógica que segue o método equals da classe Object e que nós sobrescrevemos na aula passada.

Polimorfismo

- Vejamos outro exemplo.
- Fomos contratados para desenvolver um sistema para faculdade que controle as despesas com funcionários e professores.
- A classe Funcionario pode ser algo parecido com o seguinte:

Polimorfismo

```
public class Funcionario {  
    private String nome;  
    private double salario;  
  
    public double getGastos() {  
        return this.salario;  
    }  
  
    public String getInfo() {  
        return "nome: " + this.nome + " com salário " + this.salario;  
    }  
  
    //getters & setters
```

Polimorfismo

- Já a classe Professor seria um pouco diferente, mas com a mesma base de Funcionario.
 - Afinal, professor é um funcionário da faculdade.
- O gasto que temos com o professor não é apenas seu salário.
 - Temos de somar um bônus de 10 reais por hora/aula.
 - Teremos de sobrescrever os métodos.
- A classe Professor pode ser algo parecido com o seguinte:

```
public class Professor extends Funcionario {
    private int horasDeAula;

    public double getGastos() {
        return this.getSalario() + this.horasDeAula * 10;
    }

    public String getInfo() {
        String informacaoBasica = super.getInfo();
        String informacao = informacaoBasica + " horas de aula: "
                                   + this.horasDeAula;

        return informacao;
    }

    public int getHorasDeAula() {
        return horasDeAula;
    }

    public void setHorasDeAula(int horasDeAula) {
        this.horasDeAula = horasDeAula;
    }
}
```

Polimorfismo

- Agora imagine que precisamos construir uma classe para geração de relatório.

```
public class GeradorRelatorio {  
    public void adiciona(Funcionario f) {  
        System.out.println(f.getInfo());  
        System.out.println(f.getGastos());  
    }  
}
```

- Dessa forma, podemos passar para nossa classe qualquer variável de tipo Funcionario.
- Funciona tanto para funcionário quanto para professor.

Polimorfismo

- Vamos supor que no futuro chegue uma tarefa para criar uma nova classe que representa um reitor.
- Reitor também será um funcionário da faculdade, então a classe Reitor herdará da classe Funcionario.

```
public class Reitor extends Funcionario {  
    public String getInfo() {  
        return super.getInfo() + " e ele é um reitor";  
    }  
}
```

- Será que precisaremos alterar algo na nossa classe GeradorRelatorio?

Polimorfismo

- Não!
- Quando programamos a classe GeradorDeRelatorio nunca imaginamos que existiria uma classe Reitor e, mesmo assim, o sistema funciona, graças ao polimorfismo.

Polimorfismo

- Em outras palavras, polimorfismo permite programar “no geral” em vez de “no específico”, pois permite que nossos programas manipulem objetos que herdam de uma mesma superclasse como se fossem objetos da própria superclasse.
- Possibilita projetar e implementar sistemas mais extensíveis.
 - Novos comportamentos podem ser adicionados com pouca ou nenhuma modificação nas partes gerais do programa, contanto que as novas classes sejam parte da hierarquia de herança que o programa processa genericamente.

Polimorfismo

- Quando uma variável da superclasse contém uma referência a um objeto da subclasse e essa referência é utilizada para chamar um método, o método da subclasse é chamado.
 - Isso é possível porque um objeto da subclasse é um objeto da superclasse.
 - Vale notar que o oposto não é verdade.

Polimorfismo

- Ao encontrar uma chamada de método por meio de uma variável, o compilador determina se o método pode ser chamado verificando o tipo da variável.
 - Se o tipo for uma classe que contém a declaração do método direta ou indiretamente (por herança), a chamada é compilada.
- A verificação do tipo de objeto que uma variável referência é determinado em tempo de execução.
 - Isso interfere na escolha do método que será executado quando este é sobrescrito.
 - Este processo é chamado de **vinculação dinâmica**.
 - ou dynamic binding.

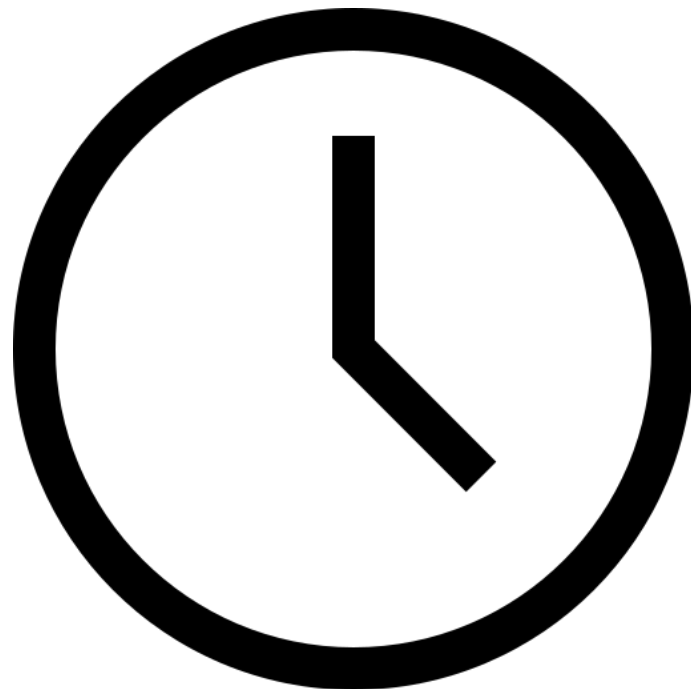
INTERVALO!

Finalizamos o nosso primeiro período de hoje. Que tal descansar um pouco?!

Nos vemos em 20 minutos.

Início: 20:15

Retorno: 20:35




instanceof e downcasting

- Voltando ao exemplo do sistema de uma faculdade.
- E se no método “adiciona”, ou em algum outro que também recebe referência a Funcionario, eu precisar invocar um método que só exista em alguma subclasse específica?
- Vamos supor que a classe Professor possua um método “getQtdDisciplinas” que retorne a quantidade de disciplinas que este professor vai ministrar no semestre.

instanceof e downcasting

```
public class GeradorRelatorio {  
    public static void adiciona(Funcionario f) {  
        System.out.println(f.getGastos());  
        System.out.println(f.getInfo());  
        System.out.println(f.getQtdDisciplinas());  
    }  
}
```

 The method getQtdDisciplinas() is undefined for the type Funcionario

instanceof e downcasting

- Não conseguimos invocar a partir de uma referência a Funcionario, pois a classe Funcionario não possui esse método.
- Como podemos fazer isso, sem precisar criar outro método específico para a subclasse Professor?
- Podemos verificar a classe de um objeto com o operador **instanceof**, que retorna “true” se o objeto à esquerda do operador é uma instância da classe à direita do operador em tempo de execução.

instanceof e downcasting

```
public static void adiciona(Funcionario f) {  
    System.out.println(f.getGastos());  
    System.out.println(f.getInfo());  
    if (f instanceof Professor) {  
        System.out.println(f.getQtdDisciplinas());  
    }  
}
```

instanceof e downcasting

- Mas o problema segue, pois a variável “f” ainda faz referência a Funcionario, que não possui esse método.
- Atribuir uma variável da subclasse a uma variável da superclasse é simples e direto.
 - Todo o objeto da subclasse é um objeto da superclasse.
 - Comportamento similar a coerção automática na promoção de tipos primitivos.
- Mas para atribuir uma variável da superclasse a uma variável da subclasse, é necessário realizar uma **coerção explícita**.
 - Também chamado de **downcasting**.

instanceof e downcasting

```
public static void adiciona(Funcionario f) {  
    System.out.println(f.getGastos());  
    System.out.println(f.getInfo());  
    if (f instanceof Professor) {  
        Professor prof = (Professor) f; //downcasting  
        System.out.println(prof.getQtdDisciplinas());  
    }  
}
```

Material complementar

- Encapsulamento, Polimorfismo, Herança em Java:
<https://www.devmedia.com.br/encapsulamento-polimorfismo-heranca-em-java/12991>
- O que é polimorfismo:
http://www.dsc.ufcg.edu.br/~jacques/cursos/p2/html/oo/o_que_e_polimorfismo.htm

<LAB365>

Interface

AGENDA

- Interface

Interface

- Vamos utilizar nossos conhecimentos adquiridos até o momento para expandir um pouco nosso exemplo de um sistema bancário.
- Atualmente temos algumas classes relacionadas a Contas e Cliente.

Interface

- Temos também uma classe ContaInvestimentos, que também é uma Conta.

```
public class ContaInvestimentos extends Conta {  
  
    public ContaInvestimentos(String nomeTitular,  
                               String sobrenomeTitular, String cpfTitular) {  
        super(nomeTitular, sobrenomeTitular, cpfTitular);  
    }  
}
```

- Vamos supor que precisamos implementar uma nova funcionalidade para tributar alguns bens dos nossos clientes.
 - ContaPoupanca não é tributável.
 - ContaCorrente e ContaInvestimentos, sim.

Interface

- ContaCorrente precisa pagar 1% do saldo.
- Então vamos criar um método getValorImposto() nessa classe, para realizar esse cálculo.

```
public class ContaCorrente extends Conta {  
    // ...  
    public double getValorImposto() {  
        return super.getSaldo() * 0.01;  
    }  
    // ...  
}
```

Interface

- ContaInvestimentos precisa pagar 2% do saldo.
- Então vamos criar o método getValorImposto() nessa classe também.

```
public class ContaInvestimentos extends Conta {  
    // ...  
    public double getValorImposto() {  
        return super.getSaldo() * 0.02;  
    }  
    // ...  
}
```

Interface

- Reparem que não criamos esse método na superclasse Conta, pois não são todos os tipos de conta que precisam pagar imposto.
- Reparem também que o cálculo do imposto muda de acordo com o tipo de Conta.

Interface

- Vamos criar uma nova classe ManipuladorDeTributaveis, contendo um atributo "total" de tipo double e um método calculaImpostos().
- Esse método não deve retornar nada, e deve receber como argumento uma lista de contas para calcular o imposto de cada uma e somar ao atributo "total".

Interface

- Mas aí nos deparamos com um problema: não podemos receber variáveis do tipo Conta para utilizar o método getValorImposto, pois a classe Conta não possui esse método.
- Como podemos resolver esse problema?

```
public class ManipuladorDeTributaveis {  
    private double total;  
  
    public void calculaImpostos(Conta[] contas) {  
        for (Conta conta : contas) {  
            this.total += conta.getValorImposto();  
        }  
    }  
}
```

Interface

- Uma possibilidade é utilizar a sobrecarga de métodos, criando dois métodos “calculaImpostos”, com parâmetros diferentes.

```
public void calculaImpostos(ContaCorrente[] contas) {  
    for (ContaCorrente conta : contas) {  
        this.total += conta.getValorImposto();  
    }  
}  
  
public void calculaImpostos(ContaInvestimentos[] contas) {  
    for (ContaInvestimentos conta : contas) {  
        this.total += conta.getValorImposto();  
    }  
}
```

Interface

- Mas já vimos que essa não é a melhor saída.
- Cada vez que criarmos uma nova subclasse de Conta que é tributável, precisaríamos adicionar um novo método de calculaImpostos no ManipuladorDeTributaveis.

Interface

- Uma solução mais interessante poderia ser criar uma nova classe no meio da árvore atual.
- Seria uma subclasse direta de Conta, chamada ContaTributavel, e fazer todos os tipos de conta tributáveis herdarem dessa nova classe.
 - Essa nova classe intermediária pode ser abstrata, inclusive.
 - Pode declarar o método getValorImposto como abstrato, forçando as subclasses a implementarem, cada uma, sua regra de valor de imposto.
- Então o método calculaImpostos no ManipuladorDeTributaveis poderia receber como argumento uma lista de ContaTributavel.

Interface

```
public abstract class ContaTributavel extends Conta {  
    public ContaTributavel(String nomeTitular,  
        String sobrenomeTitular, String cpfTitular) {  
        super(nomeTitular, sobrenomeTitular, cpfTitular);  
    }  
  
    public abstract double getValorImposto();  
}
```

Interface

```
public class ContaCorrente extends ContaTributavel

public class ContaInvestimentos extends ContaTributavel

public class ManipuladorDeTributaveis {
    private double total;

    public void calculaImpostos(ContaTributavel[] contas) {
        for (ContaTributavel conta : contas) {
            this.total += conta.getValorImposto();
        }
    }
}
```

Interface

- Herança resolveu esse caso.
- Mas e se a gente quiser criar um novo tipo de produto financeiro, que também é tributável, mas não se encaixa como um tipo de Conta?
- Por exemplo, teremos agora uma classe SeguroDeVida, que tem uma taxa fixa de 42 reais, mais 2% do valor do seguro.

Interface

```
public class SeguroDeVida {  
    private double valor;  
    private Cliente titular;  
    private int numeroApolice;  
    // ...  
    public double getValorImposto() {  
        return 42 + (this.valor * 0.02);  
    }  
    // ...  
}
```

Interface

- E como passamos um objeto SeguroDeVida para o método calculaImpostos, se ele não é uma ContaTributavel?
- Criar um novo método calculaImpostos nós já vimos que não é a melhor solução.
- Fazer da classe SeguroDeVida uma subclasse da ContaTributavel resolveria nosso problema, certo?

Interface

- Mas é uma herança sem sentido, pois SeguroDeVida definitivamente não é uma ContaTributavel.
- Se fizermos isso, SeguroDeVida terá um método “sacar”, outro “transferir”, e outros métodos e atributos que não fazem sentido existir para essa classe.
- Não devemos fazer herança quando a relação não é estritamente “é um”.

Interface

- Mas como resolver esse problema então?
- Precisamos arranjar um jeito para que essas classes garantam a existência do método “getValorImposto”, como uma espécie de contrato entre elas.

Interface

- Toda classe define 2 itens:
 - O que uma classe faz (as assinaturas dos métodos)
 - Como uma classe faz essas tarefas (o corpo dos métodos e atributos privados)
- Precisamos de um contrato que define o primeiro, ou seja, tudo o que uma classe deve fazer se quiser ter o status “Tributável”.
- Algo como:
 - contrato Tributavel:
 - quem quiser ser Tributavel precisa saber fazer:
 - 1.calcular seus impostos, devolvendo um double

Interface

- Quem quiser, pode "assinar" esse contrato, sendo assim obrigado a explicar como será feito o cálculo da tributação.
- A vantagem é que, se uma ContaCorrente "assinar" esse contrato, podemos nos referenciar a uma ContaCorrente como um Tributavel.
- Fazemos esse contrato em Java através de **interfaces**.

Interface

```
public interface Tributavel {  
    double getValorImposto();  
}
```

Interface

- Chama-se interface pois é a forma pela qual outras classes podem conversar com um Tributavel.
 - Interface é a maneira através da qual conversamos com um objeto.
- Lemos a interface da seguinte maneira: "quem desejar ser tributável precisa saber calcular seus impostos, retornando um double".
- Ela é um contrato onde quem assina se responsabiliza por implementar esses métodos (cumprir o contrato).

Interface

- Uma interface pode definir uma série de métodos, mas nunca conter implementação deles.
- Ela só expõe **o que** o objeto deve fazer, e não como ele faz, nem o que ele tem.
- **Como** ele faz, vai ser definido em uma implementação dessa interface.

Interface

- A ContaCorrente pode “assinar” esse contrato, ou seja, implementar essa interface.
- Assim que ela implementa essa interface, ela é obrigada a escrever os métodos exigidos pela interface.
- Métodos de uma interface são sempre públicos e abstratos.
- Atributos de uma interface são sempre public static final.

Interface

- Para uma classe implementar uma interface, utiliza-se a palavra-chave “implements” na assinatura da classe.

```
public class ContaCorrente extends Conta implements Tributavel
```

- A classe ContaCorrente volta a estender a classe Conta diretamente, pois substituímos a classe intermediária pela interface.
- A partir de agora, podemos tratar uma ContaCorrente como um Tributavel.
 - Assim, ganhamos comportamento polimórfico e temos mais uma forma de referenciar uma ContaCorrente.

Interface

- Quando criamos uma variável do tipo Tributavel, estamos criando uma referência para qualquer objeto de qualquer classe que implemente Tributavel, direta ou indiretamente.

```
public static void main(String[] args) {  
    Tributavel t = new ContaCorrente();  
}
```


Interface

- Sendo assim, podemos alterar o método calculaImpostos da classe ManipuladorDeTributaveis para ter como parâmetro uma lista de objetos do tipo Tributavel.

```
public void calculaImpostos(Tributavel[] bens) {  
    for (Tributavel bem : bens) {  
        this.total += bem.getValorImposto();  
    }  
}
```

Interface

- Podemos excluir a classe intermediária ContaTributavel.
- As classes que foram tributáveis devem implementar essa interface, então podemos definir isso nas classes ContaInvestimento e SeguroDeVida.
- Dessa forma, podemos passar objetos de qualquer uma dessas 3 classes para o método calculaImpostos.
- O método não faz a menor ideia de que tipo de objeto está sendo passado, mas ele tem a garantia de que esse objeto possui o método “getValorImposto”.

Interface

- Quando tivermos mais um produto financeiro que deve ser tributado, basta que ele implemente essa interface, e já vai poder ser passado como argumento ao método de cálculo de impostos.
- Pouco importa quem o objeto referenciado realmente é, pois ele tem um método `getValorImposto` que é o necessário para nosso `ManipuladorDeTributaveis` funcionar corretamente.

Interface

- Se a partir de amanhã for definido que ContaPoupanca deve ser tributada também, basta ela implementar a interface Tributavel e já pode ser utilizada no método calculaImpostos.
- Veja o desacoplamento que isso nos proporciona:
 - Cada classe que implementa a interface tem apenas essa única relação de precisar implementar o método que o “contrato” exige.
 - Cada uma pode implementar do seu jeito, contanto que o implemente.
 - Quem escreveu o Manipulador de Tributáveis não sabe que objeto receberá, só precisa saber que será um Tributavel.

Interface

- Reforçando: a interface define que todos vão saber calcular seus impostos (o que faz), enquanto a implementação define como exatamente vai ser feito esse cálculo (como faz).
- A maneira como os objetos se comunicam num sistema orientado a objetos é muito mais importante do que como eles executam.
- O que um objeto faz é mais importante do que como ele faz.
- Aqueles que seguem essa regra, terão sistemas mais fáceis de manter e modificar.

Interface

- Vimos em herança que uma classe só pode estender uma única superclasse.
- No caso de interfaces, uma classe pode implementar mais de uma interface ao mesmo tempo

```
public class ContaCorrente extends Conta implements Tributavel, Teste
```

Interface

- Diferentemente das classes, uma interface pode herdar de mais de uma interface.
- É como um contrato que depende que outros contratos sejam fechados antes deste valer.
- Você não herda métodos e atributos, mas sim responsabilidades.

```
public interface Teste extends Tributavel, OutroTeste
```

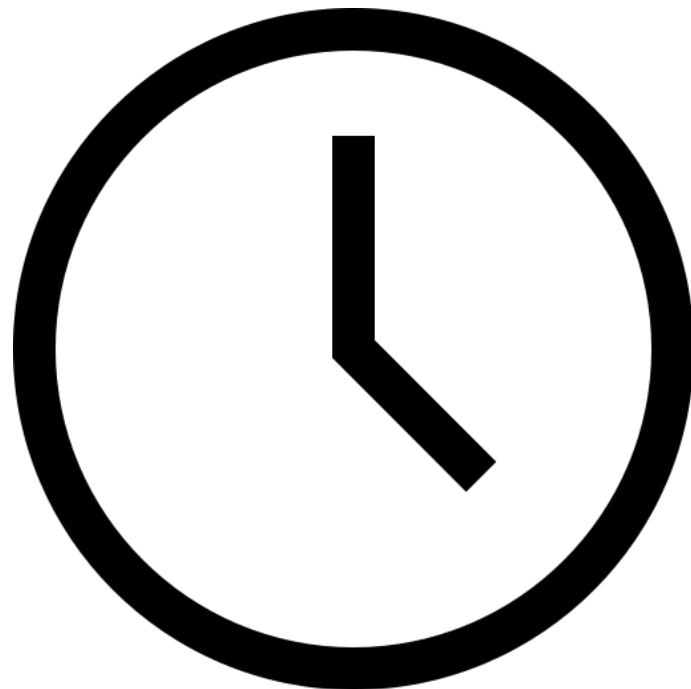
INTERVALO!

Finalizamos o nosso primeiro período de hoje. Que tal descansar um pouco?!

Nos vemos em 20 minutos.

Início: 20:20

Retorno: 20:40



Material complementar

- Interfaces em Java:
<https://www.devmedia.com.br/entendendo-interfaces-em-java/25502>
- Métodos estáticos e default em interfaces:
<https://www.baeldung.com/java-static-default-methods>
- Interface funcional:
<https://pt.stackoverflow.com/questions/11162/o-que-s%C3%A3o-interfaces-funcionais>

<LAB365>

Revisão de POO

AGENDA

- Reforço de POO:
 - Classe e objeto
 - Encapsulamento
 - Herança
 - Interface
 - Polimorfismo

Revisão POO

- Relembrando:
- Uma classe é um conjunto de características e comportamentos que definem o conjunto de objetos pertencentes à essa classe.
- A classe em si é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto.
- Chamamos essa criação de instanciação da classe, como se estivéssemos usando esse molde (classe) para criar um objeto.

Revisão POO

- Vamos considerar uma classe Carro.
- Carro contém atributos como: modelo, velocidade, cor, quantidade de portas, etc.
- Um carro também possui comportamentos, realiza operações, ações, que são seus métodos, como: acelerar, desacelerar, acender os faróis, buzinar e tocar música.

```
public class Carro {
    String modelo;
    double velocidade;
    String cor;
    int qtdPortas;

    public Carro(String modelo, double velocidade, String cor, int qtdPortas) {
        this.modelo = modelo;
        this.velocidade = velocidade;
        this.cor = cor;
        this.qtdPortas = qtdPortas;
    }

    public void acelerar() {
        /* código do carro para acelerar */
    }

    public void frear() {
        /* código do carro para frear */
    }

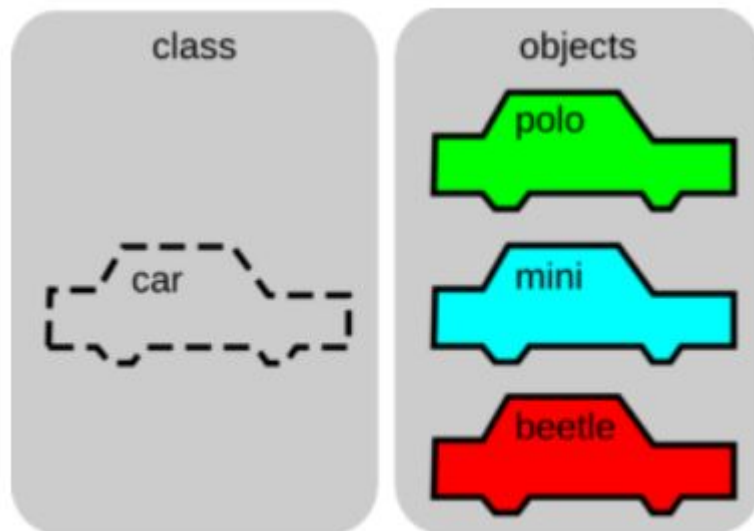
    public void acenderFarol() {
        /* código do carro para acender o farol */
    }

    public void buzinar() {
        /* código do carro para buzinar */
    }
}
```

Revisão POO

- Seu carro é um objeto seu, mas na loja onde você o comprou existiam vários outros, muito similares: com quatro rodas, volante, câmbio, retrovisores, faróis, dentre outras partes.
- Observe que, apesar do seu carro ser único (por exemplo, possui um registro único no DETRAN), podem existir outros com exatamente os mesmos atributos, ou parecidos, ou mesmo totalmente diferentes, mas que ainda são considerados carros.
- Podemos dizer então que seu objeto pode ser classificado (isto é, seu objeto pertence à uma classe) como um carro, e que seu carro nada mais é que uma instância dessa classe chamada "carro".

Revisão POO



Fonte: [POO: o que é programação orientada a objetos?](#)

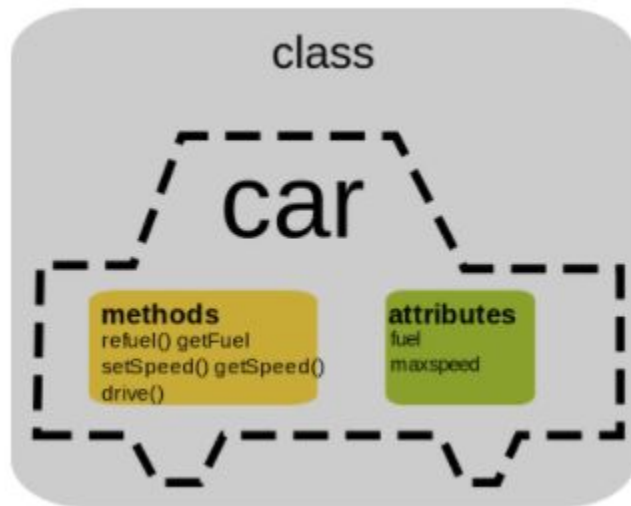
Encapsulamento

- As duas bases da POO são os conceitos de classe e objeto.
- Desses conceitos, derivam alguns outros conceitos extremamente importantes ao paradigma, que não só o definem, como também são as soluções de alguns problemas da programação estruturada.
- Esses conceitos são: o encapsulamento, a herança, as interfaces e o polimorfismo.

Encapsulamento

- Vimos que nossa classe Carro possui alguns atributos e alguns métodos.
- Os métodos podem utilizar os atributos, e também os outros métodos.
- Por exemplo: acelerar pode utilizar um atributo velocidade, e também invocar outro método para diminuir a quantidade de gasolina no tanque.

Encapsulamento



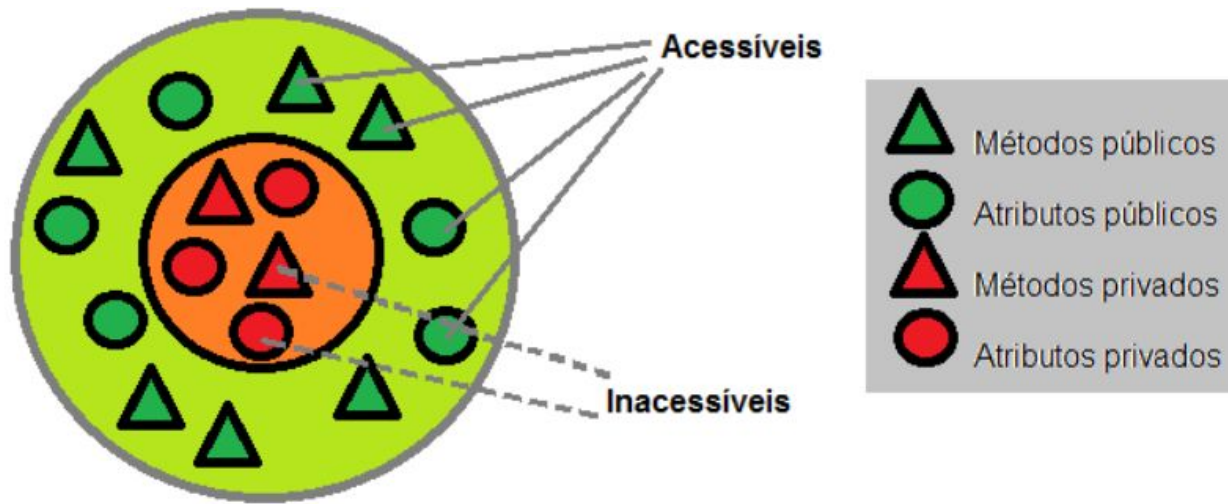
Fonte: [POO: o que é programação orientada a objetos?](#)

Encapsulamento

- Mas se alguns desses atributos e métodos forem facilmente visíveis e modificáveis, pode resultar em efeitos colaterais imprevisíveis.
- Nessa analogia, uma pessoa pode não estar satisfeita com a aceleração do carro e modifica a forma como ela ocorre, criando efeitos colaterais que podem fazer o carro nem andar, por exemplo.
- Podemos dizer, nesse caso, que o método de aceleração não deve ser visível de fora do próprio carro.

Encapsulamento

- Em POO, um atributo ou método que não é visível de fora do próprio objeto é chamado de "privado" e quando é visível, é chamado de "público".



Fonte: [POO: o que é programação orientada a objetos?](#)

Encapsulamento

- Então, como sabemos como o carro acelera?
- Não sabemos. Não temos porquê saber.
- Nós só sabemos que para acelerar, devemos pisar no acelerador e de resto o objeto sabe como executar essa ação sem expor como o faz.
- Dizemos que a aceleração do carro está encapsulada, pois sabemos o que ele vai fazer ao executarmos esse método, mas não sabemos como.
 - E na verdade, não importa para o programa como o objeto o faz, só que ele o faça.

Encapsulamento

- O mesmo vale para atributos.
- Não sabemos como o carro sabe qual velocidade mostrar no velocímetro ou como ele calcula sua velocidade.
- Mas não precisamos saber como isso é feito.
- Só precisamos saber que ele vai nos dar a velocidade certa.
- Ler ou alterar um atributo encapsulado pode ser feito a partir de getters e setters.

Encapsulamento

- Esse encapsulamento de atributos e métodos impede o chamado vazamento de escopo, onde um atributo ou método é visível por alguém que não deveria vê-lo, como outro objeto ou classe.
- Isso evita a confusão do uso de variáveis globais no programa, deixando mais fácil de identificar em qual estado cada variável vai estar a cada momento do programa.
- Já que a restrição de acesso nos permite identificar quem consegue modificá-la.

Encapsulamento

```
public class Carro {  
    private String modelo;  
    private double velocidade;  
    private String cor;  
    private int qtdPortas;  
    private MecanismoAceleracao mecanismoAceleracao;  
  
    /*  
    * Repare que o mecanismo de aceleração é inserido no carro ao ser construído, e  
    * não o vemos nem podemos modificá-lo, isto é, não tem getter nem setter. Já  
    * "modelo" pode ser visto, mas não alterado.  
    */  
    public Carro(String modelo, MecanismoAceleracao mecanismoAceleracao, String cor, int qtdPortas) {  
        this.modelo = modelo;  
        this.mecanismoAceleracao = mecanismoAceleracao;  
        this.velocidade = 0;  
        this.cor = cor;  
        this.qtdPortas = qtdPortas;  
    }  
  
    public String getModelo() {  
        return this.modelo;  
    }  
}
```

Encapsulamento

```
public double getVelocidade() {  
    return this.velocidade;  
}  
  
private void setVelocidade(double novaVelocidade) {  
    /* código para alterar a velocidade do carro  
    *  
    * Como só o próprio carro deve calcular a velocidade, esse método não pode ser  
    * chamado de fora, por isso é "private"  
    */  
}  
  
public void acelerar() {  
    double novaVelocidade = this.mecanismoAceleracao.acelerar(this.getVelocidade());  
    this.setVelocidade(novaVelocidade);  
}  
  
public String getCor() {  
    return this.cor;  
}  
  
/* podemos mudar a cor do carro quando quisermos */  
public void setCor(String cor) {  
    this.cor = cor;  
}
```

Herança

- Apesar de cada carro ser único, existem carros com exatamente os mesmos atributos ou formas modificadas.
- Digamos que você tenha comprado o modelo Fit, da Honda.
- Esse modelo possui uma outra versão, chamada WR-V (ou "Honda Fit Cross Style").
- Esse possui muitos dos atributos da versão clássica, mas com algumas diferenças bem grandes para transitar em estradas de terra:
 - o motor é híbrido (aceita álcool e gasolina)
 - possui um sistema de suspensão diferente
 - sistema de tração nas 4 rodas

Herança

- Vemos então que não só alguns atributos como também alguns mecanismos (ou métodos, traduzindo para POO) mudam.
- Mas essa versão "cross" ainda é do modelo Honda Fit, ou melhor, é um tipo do modelo.
- Quando dizemos que uma classe A é um tipo de classe B, dizemos que a classe A herda as características da classe B e que a classe B é mãe da classe A.
- Estabelecemos uma relação de herança entre elas.

Herança

- No caso do carro, dizemos então que um Honda Fit "Cross" é um tipo de Honda Fit.
- O que muda são alguns atributos (paralama reforçado, altura da suspensão etc).
- E um dos métodos da classe (acelerar, pois agora há tração nas quatro rodas).
- Mas todo o resto permanece o mesmo, e o novo modelo recebe os mesmos atributos e métodos do modelo clássico.

Herança

```
// "extends" estabelece a relação de herança com a classe Carro
public class HondaFit extends Carro {

    public HondaFit(MecanismoAceleracao mecanismoAceleracao) {
        // chama o construtor da classe mãe, ou seja, da classe "Carro"
        super("Honda Fit", mecanismoAceleracao);
    }
}
```

Interface

- Muitos dos métodos dos carros são comuns em vários automóveis.
- Tanto um carro quanto uma motocicleta são classes cujos objetos podem acelerar, parar, acender o farol etc, pois são coisas comuns a automóveis.
- Podemos dizer, então, que ambas as classes "carro" e "motocicleta" são "automóveis".

Interface

- Quando duas (ou mais) classes possuem comportamentos comuns que podem ser separados em uma outra classe, dizemos que a "classe comum" é uma interface, que pode ser "herdada" pelas outras classes.
- Notem as aspas em "classe comum", que pode ser "herdada".

Interface

- Porque uma interface não é exatamente uma classe, mas sim um conjunto de métodos.
- Métodos estes que todas as classes que "herdarem" dela devem possuir (implementar).
- Portanto, uma interface não é "herdada" por uma classe, mas sim implementada.

Interface

- No mundo do desenvolvimento de software, dizemos que uma interface é um "contrato":
- Uma classe que implementa uma interface deve fornecer uma implementação a todos os métodos que a interface define.
- E em compensação, a classe implementadora pode dizer que ela é do tipo da interface.

Interface

- No nosso exemplo, "carro" e "motocicleta" são classes que implementam os métodos da interface "automóvel".
- Logo, podemos dizer que qualquer objeto dessas duas primeiras classes, como um Honda Fit ou uma motocicleta da Yamaha, são automóveis.

Interface

- Um pequeno detalhe: uma interface não pode ser herdada por uma classe, mas sim implementada.
- No entanto, uma interface pode herdar de outra interface, criando uma hierarquia de interfaces.

Interface

- Usando um exemplo completo com carros, dizemos que:
 - A classe "Honda Fit Cross" herda da classe "Honda Fit" que por sua vez herda da classe "Carro".
 - A classe "Carro" implementa a interface "Automóvel" que, por sua vez, pode herdar uma interface chamada "MeioDeTransporte".
 - Tanto um "automóvel" quanto uma "carroça" são meios de transporte, ainda que uma carroça não seja um automóvel.

Interface

```
public interface Automovel {  
    void acelerar();  
    void frear();  
    void acenderFarol();  
}
```

Interface

```
public class Carro implements Automovel {
    /* ... */

    @Override
    public void acelerar() {
        double novaVelocidade = this.mecanismoAceleracao.acelerar(this.getVelocidade());
        this.setVelocidade(novaVelocidade);
    }

    @Override
    public void frear() {
        /* código do carro para frear */
    }

    @Override
    public void acenderFarol() {
        /* código do carro para acender o farol */
    }

    /* ... */
}
```

Interface

```
public class Moto implements Automovel {  
    /* ... */  
  
    @Override  
    public void acelerar() {  
        /* código específico da moto para acelerar */  
    }  
  
    @Override  
    public void frear() {  
        /* código específico da moto para frear */  
    }  
  
    @Override  
    public void acenderFarol() {  
        /* código específico da moto para acender o farol */  
    }  
  
    /* ... */  
}
```


Polimorfismo

- Vamos dizer que um dos motivos de você ter comprado um carro foi a qualidade do sistema de som dele.
- Mas, no seu caso, digamos que a reprodução só pode ser feita via rádio ou bluetooth, enquanto que no seu antigo carro, podia ser feita apenas via cartão SD e pendrive.
- Em ambos os carros está presente o método "tocar música", mas como o sistema de som deles é diferente, a forma como o carro toca as músicas é diferente.

Polimorfismo

- Dizemos que o método "tocar música" é uma forma de polimorfismo.
- Pois dois objetos, de duas classes diferentes, têm um mesmo método que é implementado de formas diferentes.
- Ou seja, um método possui várias formas, várias implementações diferentes em classes diferentes, mas que possuem o mesmo efeito.
- "polimorfismo" vem do grego poli = muitas, morphos = forma.

```
public static void main(String[] args) {  
  
    Automovel moto = new Moto("Yamaha XPTO-100",  
                               new MecanismoAceleracaoMoto());  
    Automovel carro = new Carro("Honda Fit",  
                                 new MecanismoAceleracaoCarro());  
    Automovel[] listaAutomoveis = new Automovel[2];  
    listaAutomoveis[0] = moto;  
    listaAutomoveis[1] = carro;  
  
    for (Automovel automovel : listaAutomoveis) {  
        automovel.acelerar();  
        automovel.acenderFarol();  
    }  
}
```

- Apesar de serem objetos diferentes, moto e carro possuem os mesmos métodos “acelerar” e “acenderFarol”, que são chamados do mesmo jeito, apesar de serem implementados de forma diferente.

Material complementar

- “O que é programação orientada a objetos?”:
<https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos>
 - (contém exemplos em Java e Python)

AVALIAÇÃO DOCENTE

O que você está achando das minhas aulas neste conteúdo?

Clique [aqui](#) ou escaneie o QRCode ao lado para avaliar minha aula.

Sinta-se à vontade para fornecer uma avaliação sempre que achar necessário.



<LAB365>

<LAB365>

SENAI