O conceito de herança na Orientação a Objeto.

O conceito de herança dentro da O.O consiste em criar novas classes aproveitando estruturas (atributos e métodos) de classes já existentes, chamamos isso de reaproveitamento de código, onde uma classe “empresta” certas estruturas para outra. A classe que pega “emprestado” essas estruturas pode por sua vez criar suas próprias estruturas se especializando ainda mais em relação a classe que “emprestou”.

A aplicação de tal conceito permite desenvolver softwares mais eficientes no que se refere a implementação de código, pois se reaproveita partes de código que já foram devidamente testadas e depuradas

A classe que já existia e é herdada é chamada de “**superclasse**” enquanto a nova classe que é criada a herdeira é chamada de “**subclasse**”, essa também pode ser chamada de “**subclasse**” ou de “**especialização**”

O relacionamento entre classes herdadas e classes herdeiras pode ser divido em duas categorias “**direto**” e “**indireto**”.

Direto – Uma nova classe chamada de “**B**” herda uma classe já existente chamada de “**A**”, logo “**A**” é uma superclasse direta de “**B**”.

Indireto - Uma nova classe chamada de “**C**” herda uma classe já existente chamada de “**B**”, **“B**” por sua vez já herda outra classe chamada de “**A**”, logo “**B**” é uma superclasse direta de “**C**” e “**A**” é uma superclasse direta de “**B**” e indireta de “**C**”.

A visibilidade protected.

A visibilidade “**protected**” está ligado diretamente ao conceito de herança esse nível define que os atributos e métodos dentro de sua definição só podem ser acessados pela própria classe que os implementou e por suas “**subclasses**” (diretas ou indiretas), sendo assim um nível de encapsulamento intermediário entre o “**public**” e o “**private**”.

Observação: Cabe aqui uma pequena revisão, atributos e métodos definidos com visibilidade “**public**” são acessíveis a todos objetos da classe que os implementou e a suas herdeiras “**subclasses**” enquanto atributos e métodos definidos como “**private**” são acessíveis apenas para a classe que os implementou e apenas por meio de acessadores que são métodos (“**getters**” e “**setters**”) criados especialmente para acessar os atributos “**private**” dessa classe ficando assim ocultos para qualquer outra classe inclusive “**subclasses**”.

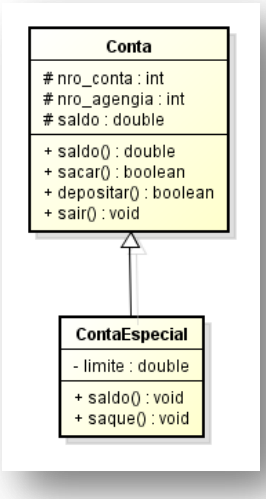
O polimorfismo.

O conceito de “**polimorfismo**” está diretamente ligado ao conceito de herança e consiste em uma relação entre os métodos da “**superclasse**” e suas “**subclasses**”. Por exemplo, imagine a nossa classe “**Conta**” que possui o método “**saldo()**”, quando esse método é chamado é exibido o valor do saldo disponível na conta naquele momento, agora imagine que criamos uma nova classe chamada “**ContaEspecial**” e essa é uma subclasse de “Conta”, logo possui seus atributos e métodos inclusive o método “**saldo()**”, porem a nova classe “**ContaEspecial**” possui um atributo a mais que é chamado de “**limite**” que define um valor além do saldo que o cliente pode sacar e que deverá ser ressarcido ao banco posteriormente, quando um cliente de “**ContaEspecial**” consulta seu saldo além do valor em conta deverá aparecer também o seu limite e é ai que está o conceito o polimorfismo (**poli = muitas**) e (**morfismo = formas**) a forma como o método saldo se comporta (implementação) na classe “**ContaEspecial**” é parecido com o da classe “**Conta**”, mas traz uma informação a mais o limite, logo poderíamos reaproveitar parte do método “**saldo()**” de “**Conta**” pois a herança permite isso e adicionar mais uma parte que seria a exibição do valor de limite disponível ao nosso cliente naquele momento.

Aplicação do conceito de herança e polimorfismo no sistema de caixa eletrônico.

Vamos agora alterar nosso projeto do caixa eletrônico para aplicar os conceitos de herança e de polimorfismo visto até aqui.

A imagem abaixo é um diagrama de classes atualizado:



Foram feitas as seguintes mudanças:

Classe “Conta”:

- A visibilidade de todos os métodos foi alterada para “**protected**” isso garante o acesso da própria classe e de suas subclasses, lembre-se

isso é feito através dos métodos acessadores de “**get**” e “**set”** que agora também serão usados pela classe “**ContaEspecial**”.

Classe “ContaEspecial”:

- A classe possui um atributo chama “**limite**” do tipo “**private**”, ou seja somente a própria classe vai poder manipular esse atributo através de seus métodos acessadores;
- O método saque foi sobrescrito dentro da classe pois sua implementação mudou isso torna o método “**saque**” de “**ContaEspecial**” um método polimórfico.
- O método saldo também foi sobrescrito dentro da classe pela mesma razão do anterior o que também torna saldo um método polimórfico.

A imagem abaixo ilustra o código da nova classe “**ContaEspecial**”:

```java
package utilitarios;

public class ContaEspecial extends Conta {

    private double limite = 500;

    @Override
    public void sacar(double valor) {

        if (valor <= super.getSaldo()) {

            super.sacar(valor);

        } else if (valor <= (this.getLimite() + super.getSaldo())) {

            valor = valor - super.getSaldo();
            super.setSaldo(0);

            this.setLimite(this.getLimite() - valor);

        }

    }

    @Override
    public void saldo() {

        super.saldo();
        System.out.printf("Seu limite é %.2f\n", this.limite);

    }

    public double getLimite() {
        return limite;
    }

    public void setLimite(double limite) {
        this.limite = limite;
    }

}
```

A linha três (3) declara a classe, mas observe o uso do código “**extends Conta**” isso indica que a classe conta especial herda a classe “**Conta**”.

Da linha oito (8) a linha vinte e três (23) temos a declaração do método “**saque()**” observe que a lógica mudou um pouco o começo é igual, mas depois se torna um pouco mais complexo. Na oito (8) temos um teste baseado em uma estrutura de decisão “**if (valor <= super.getSaldo())**”, observe que a variável valor é comprada a “**getsaldo**” só que esse método vem da superclasse graças ao uso do comando “**super**” ante do nome do método, note também que se o valor que se pretende sacar for menor do que o saldo o limite vai permanecer intacto e o processo de saque é igual ao da conta comum, logo não precisamos codificar novamente basta chamar o método “**saque()**” da superclasse como foi feito na linha doze (12).

Se for o caso do valor de saque ser maior do que o saldo vamos testar se a soma de saldo e limite é menor ou igual ao valor informado pelo usuário "**else if (valor <= (this.getLimite() + super.getSaldo()))**", observe que chamamos dois métodos para realizar a soma um é "**this.getLimite()**" que pertence à classe "**ContaEspecial**" e o outro é "**super.getSaldo()**" que pertence a superclasse "**Conta**", sendo menor o teste retorna verdadeiro "**true**" o valor de saldo é zerado e limite sofre um abatimento do excedente conforme ilustra a imagem abaixo:

```
16              valor = valor - super.getSaldo();
17              super.setSaldo(0);
18
19              this.setLimite(this.getLimite() - valor);
```

A linha dezesseis (16) abate da variável "valor" o seu próprio montante menos o saldo, observe a chamada do método "**getSlado()**" da superclasse o que sobra vai ser abatido de limite;

A linha dezessete (17) zera o saldo pois o valor sacado excede seu valor de qualquer forma.

A linha dezenove (19) abate o restante do valor do limite podendo ou não chegar a zero.

Construtores.

Construtores são métodos especiais que toda classe possui por padrão dentro da JVM quando instanciamos um objeto um construtor é sempre requerido e executado, ou seja os construtores são métodos que são executados sempre que um objeto de uma classe for criado e isso independe de nossa vontade é uma característica do JAVA. Quando um programador não declara um construtor o JAVA usa o que é chamado de construtor padrão cuja a função é configurar o valor padrão para cada atributo da classe de acordo como seu tipo.

Um programador pode então declarar seu próprio construtor e implementar suas funcionalidades de acordo com a necessidade da classe em questão, assim quando houver uma instancia de objeto dessa classe o método construtor será executado automaticamente.

Os métodos construtores são declarados como mesmo nome da classe é não possuem tipo de retorno nem mesmo a palavra reservada "**void**" que indica que não há retorno.

O construtor é representado no momento da criação do objeto através dos parênteses no final da linha. Exemplo:

```
MenuConta mc = new MenuConta();
```

Um construtor pode ou não possuir parâmetros como qualquer outro método e a regra é exatamente a mesma, esse deve possuir um nome, um tipo e uma vez declarado deve ser obrigatoriamente recebido pelo método construtor quando esse for chamado (momento em que se instancia um objeto de uma classe).

Mais uma vez para demostrar a aplicação do conceito vamos realizar uma alteração em nosso aplicativo do caixa eletrônico adicionando dois menus agora.

Um menu vai permitir que o usuário escolha entre uma conta comum ou uma conta corrente, classe "**MenuConta.java**" e o outro menu vai permitir que o usuário realize as respectivas ações (saldo, saque ou depósito) em cada uma das contas, classe "**MenuOperacao.java**".

A imagem abaixo ilustra a classe "**MenuConta.java**":

```java
package menu;

import java.util.Scanner;
import utilitarios.Conta;
import utilitarios.ContaEspecial;

public class MenuConta {

    public MenuConta() {

        Scanner sc = new Scanner(System.in);
        int opc = 0;

        while (opc != 3) {

            System.out.println("Digite uma opção:");
            System.out.println("1 - Conta Comum");
            System.out.println("2 - Conta Especial");
            System.out.println("3 - Sair");

            opc = sc.nextInt();

            if (opc != 3) {

                MenuOperacao mo = new MenuOperacao(opc);

            }

        }
    }
}
```

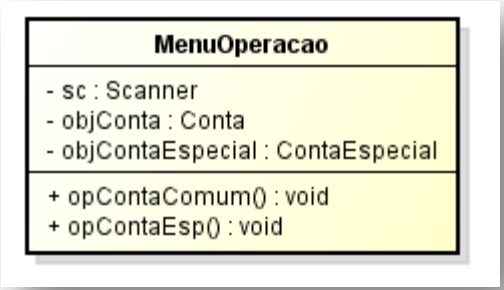
Observe que a classe possui um único método de mesmo nome e que esse não possui nenhum tipo de indicação de retorno de dados, “**void**” essas características definem esse método como um construtor da classe.

A partir daí tudo é muito parecido com a lógica do menu que existia antes na classe principal, temos um objeto da classe “**Scanner**” necessário para recuperar dados informados pelo usuário via teclado, uma variável do tipo “**int.**” que vai armazenar os dados e um loop de repetição “**while**” com uma condição que mantem o usuário dentro do “**menu**” até que esse opte por sair.

O mais importante aqui é a estrutura “**if**” que é usada para chamar instanciar um objeto da classe “**MenuOperacao**”, observe a condição do “**if**” que impede que esse objeto seja criado caso o usuário decida sair do sistema e note também que no processo de criação do objeto da classe “**MenuOperacao**” é passado um valor como parâmetro onde esse valor é o tipo da conta que o usuário que manipular.

A classe “**MenuConta.java**” é mais complexa e seu código é um tanto extenso então vou destacar apenas as partes mais importante aqui, porem você pode ver o código completo no arquivo **PDF** “**MenuOperação.pdf**” que foi entregue junto com esse material.

A imagem abaixo ilustra o diagrama de classe de “**MenuConta**”:



Observe que a classe possui três atributos senso “**sc**” que vai representar um objeto da classe “Scanner”, logo seu tipo é definido como o dessa classe, “**objConta**” que vai representar um objeto da classe “Conta” e **objContaEspecial**” que vai representar um objeto da classe “**ContaEspecial**”.

```
7   public class MenuOperacao {
8
9       Scanner sc = new Scanner(System.in);
10      Conta objConta = new Conta();
11      ContaEspecial objContaEsp = new ContaEspecial();
12
13      public MenuOperacao(int opc) {
```

Observe as linhas de nove (9) a dez (10) onde os atributos discutidos acima são declarados e os seus respectivos objetos já são criados, criar esses objetos e armazena-los em atributos ao invés de variáveis locais permite chamar e manipular esses objetos em todo o escopo da classe.

Observação: Note também que não foi usado nenhum modificar de acesso (visibilidade) quando isso ocorre o JAVA toma a visibilidade dos atributos como “**public**”.

Por fim na linha treze (13) temos a declaração do método construtor que é muito parecido com o feito anteriormente salvo o fato que o construtor da classe em questão exige um parâmetro do tipo “**int**” que vai definir qual a conta a ser acessada pelo menu.

Note que o corpo do método construtor possui uma estrutura “**if...else**” entre as linhas quinze (15) e vente e três (23) que define qual objeto vai ser instanciado baseado no valor recebido no parâmetro. Observe a imagem abaixo:

```
13      public MenuOperacao(int opc) {
14
15          if (opc == 1) {
16
18              this.opContaCommum();
19
20          } else {
22              this.opContaEsp();
23
24          }
25      }
```

Os métodos “**opContaCommum**” e “**opContaEsp**” implementam a lógica do menu de execução das operações das contas que antes estava no método “**main**” da classe principal e não há nenhuma novidade aqui.

A imagem abaixo ilustra o código do método “**opContaCommum**”:

```java
public void opContaCommum() {

    int opcopr = 0;
    double valor = 0;

    while (opcopr != 4) {

        System.out.println("Digite uma opção:");
        System.out.println("1 - Saldo");
        System.out.println("2 - Deposito");
        System.out.println("3 - Saque");
        System.out.println("4 - Sair");

        opcopr = sc.nextInt();

        switch (opcopr) {

            case 1:
                objConta.saldo();
                break;

            case 2:
                System.out.println("Digite um valor de deposito:");
                valor = sc.nextDouble();
                objConta.deposito(valor);
                objConta.saldo();
                break;

            case 3:
                System.out.println("Digite um valor de saque");
                valor = sc.nextDouble();
                objConta.sacar(valor);
                objConta.saldo();
                break;

            case 4:
                break;

            default:
                System.out.println("Opção inválida");

        }

    }

}
```

A imagem abaixo ilustra o código do método “**opContaEsp**”:

```java
public void opContaEsp() {

    int opcopr = 0;
    double valor = 0;

    while (opcopr != 4) {

        System.out.println("Digite uma opção:");
        System.out.println("1 - Saldo");
        System.out.println("2 - Deposito");
        System.out.println("3 - Saque");
        System.out.println("4 - Sair");

        opcopr = sc.nextInt();

        switch (opcopr) {

            case 1:
                objContaEsp.saldo();
                break;

            case 2:
                System.out.println("Digite um valor de deposito:");
                valor = sc.nextDouble();
                objContaEsp.deposito(valor);
                objContaEsp.saldo();
                break;

            case 3:
                System.out.println("Digite um valor de saque");
                valor = sc.nextDouble();
                objContaEsp.sacar(valor);
                objContaEsp.saldo();
                break;

            case 4:
                break;

            default:
                System.out.println("Opção inválida");

        }

    }

}
```

Note que são os mesmos códigos usados antes porem agora temos um que manipula os objetos da classe “**ContaComum**” e outro que manipula os objetos da classe “**ContaEspecial**”.

Por fim temos a imagem de como ficou nossa classe principal:

```java
package caixaeletronicosumare;

import menu.MenuConta;

public class CaixaEletronicoSumare {

    public static void main(String[] args) {

        MenuConta mc = new MenuConta();

    }

}
```

Faça uma reflexão agora, olhe a classe principal da aula anterior assim como a estrutura do projeto e compare com a de agora e você vai notar que quando aplicamos os conceitos de O.O em um uma linguagem como o JAVA que está muito próxima desses conceitos conseguimos desenvolver softwares altamente modulados que permitem uma rápida manutenção, depuração e escalabilidade.