Objetivo da Unidade*:

- **Conhecer e aplicar os conceitos mais importantes associados a orientação a objetos.**

5.1 Métodos e atributos estáticos

Atributos estáticos são atributos que contêm informações inerentes a uma classe e não a um objeto em especíico. Por isso são também conhecidos como atributos ou variáveis de classe.

Por exemplo, suponha que quiséssemos ter um atributo que indicasse a quantidade de contas criadas. Esse atributo não seria inerente a uma conta em especíico, mas a todas as contas. Assim, seria definido como um atributo estático. Para definir um atributo estático em Java, basta colocar a palavra static entre o qualiicador e o tipo do atributo.

O mesmo conceito é válido para métodos. Métodos estáticos são inerentes à classe e, por isso, não nos obrigam a instanciar um objeto para que possamos utilizá-los. Para definir um método como estático, basta utilizar a palavra static, a exemplo do que acontece com atributos. Para utilizar um método estático devo utilizar o nome da classe acompanhado pelo nome do método.

Métodos estáticos são muito utilizados em classes do Java que proveem determinados serviços. Por exemplo, Java fornece na classe Math uma série de métodos estáticos que fazem operações matemáticas como: raiz quadrada (sqrt), valor absoluto (abs), seno (sin) entre outros. A Figura 38 apresenta exemplos de uso dos métodos da classe Math a fim de ilustrar a utilização de métodos estáticos.

```
double x = Math.sqrt(634);//Atribui a x o valor da raiz quadrada de 634
double z = Math.sin(4);//Atribui a x o valor do seno de 4 radianos
double y = Math.PI; //Atribui a y o valor da constante matemática pi.
```

Figura 38: Utilização de métodos estáticos da classe Math

Note no exemplo da Figura 38 que não instanciamos objetos da classe Math para utilizar seus métodos e constantes, pois são estáticos.
Um método criado como estático só poderá acessar atributos que também sejam estáticos, além dos seus argumentos e variáveis locais.

5.2 Polimorfismo

A palavra polimorismo vem do grego poli morfos e significa muitas formas. Na orientação a objetos, isso representa uma característica que permite que classes diferentes sejam tratadas de uma mesma forma.

O polimorfismo permite escrever programas que processam objetos que compartilham a mesma superclasse (direta ou indiretamente) como se todos fossem objetos da superclasse; isso pode simplificar a programação (DEITEL; DEITEL, 2010, p. 305).

Em outras palavras, podemos ver o polimorfismo como a possibilidade de um mesmo método ser executado de forma diferente de acordo com a classe do objeto que aciona o método e com os parâmetros passados para o método.

Com o polimorfismo podemos projetar e implementar sistemas que são facilmente extensíveis – novas classes podem ser adicionadas com pouca ou nenhuma alteração a partes gerais do programa, contanto que as novas classes façam parte da hierarquia de herança que o programa processa genericamente. As únicas partes de um programa que devem ser alteradas para acomodar as novas classes são aquelas que exigem conhecimento direto das novas classes que adicionamos à hierarquia (DEITEL; DEITEL, 2010, p. 305).

O polimorfismo pode ser obtido pela utilização dos conceitos de herança, sobrecarga de métodos e sobrescrita de método (também conhecida como redefinição ou reescrita de método).

5.3 Sobrescrita

A técnica de sobrescrita permite reescrever um método em uma subclasse de forma que tenha comportamento diferente do método de mesma assinatura existente na sua superclasse.

Para ilustrar o conceito de sobrescrita, vamos criar um método imprimirTipoConta() na superclasse Conta e vamos sobrescrevê-lo nas duas subclasses para esse exemplo (ContaEspecial e ContaPoupanca). Esse método simplesmente imprimirá na tela uma mensagem de acordo com o tipo da conta, ou seja, de acordo com o tipo do objeto ele imprimirá uma mensagem diferente. A Figura 39 exibe apenas a linha de definição de cada classe e seu respectivo método imprimirTipoConta() (omitimos o resto dos códigos das classes e as juntamos todas em uma única figura por uma questão de espaço).

```
public class Conta {

    public void imprimirTipoConta(){
        System.out.println("Conta Comum");
    }
}
public class ContaEspecial extends Conta {

    @Override
    public void imprimirTipoConta(){
        System.out.println("Conta Especial");
    }
}
public class ContaPoupanca extends Conta {

    @Override
    public void imprimirTipoConta() {
        System.out.println("Conta Poupança");
    }
}
```

Figura 39: Exemplo de sobrescrita de método

Note que no exemplo da Figura 39, nas linhas anteriores aos métodos ImprimirTipoConta das classes ContaEspecial e ContaPoupança, há uma notação **@Override**. A notação **@Override** é inserida automaticamente pelo NetBeans para indicar que esse método foi definido no ancestral e está sendo redefinido na classe atual. A não colocação da notação **@Override** não gera erro, mas gera um aviso (Warning). Isso ocorre porque entende-se que, quando lemos uma classe e seus métodos, é importante existir alguma forma de sabermos se um certo método foi ou não definido numa classe ancestral. Assim a notação **@Override** é fundamental para aumentar a legibilidade e manutenibilidade do código. A Figura 40 exibe um exemplo de utilização dos métodos apresentados na Figura 39, a fim de ilustrar o polimorfismo.

```java
package banco;
import java.util.Scanner;

public class UsaContaPolimorfa {
    public static void main(String[] args) {
        Conta c = null;
        Scanner scan = new Scanner(System.in);
        int opcao;
        System.out.println("Qual tipo de conta deseja criar para José?");
        System.out.println("1 - Conta");
        System.out.println("2 - Conta especial");
        System.out.println("3 - Conta poupança");
        opcao = scan.nextInt();
        switch (opcao) {
            case 1:
                c = new Conta(1, "José");
                break;
            case 2:
                c = new ContaEspecial(1, "José", 100.00);
                break;
            case 3:
                c = new ContaPoupanca(1, "José");
                break;
        }
        c.imprimirTipoConta();
    }
}
```

Figura 40: Exemplo de polimorfismo

No exemplo da Figura 40, vemos que, de acordo com a opção escolhida no menu impresso, temos a criação de um objeto diferente. Quando o usuário digita 2, por exemplo, é criada uma instância de ContaEspecial. Note que a variável c é do tipo Conta. Mas, ainda assim, é possível atribuir a ela uma instância de ContaPoupanca ou de ContaEspecial pois ambas herdam da classe Conta.

Na última linha do código apresentado, o método imprimirTipoConta() é chamado. Mas, sabemos que esse método foi implementado na classe Conta e sobrescrito nas duas classes ilhas. Assim, qual das implementações será usada por essa chamada? A resposta a essa pergunta vai depender da opção digitada pelo usuário! Por exemplo, caso o usuário digite 2, a variável c receberá uma instância de ContaEspecial. Nesse caso, na última linha será chamado o método imprimirTipoConta() da classe ContaEspecial. Analogamente, caso o usuário digite a opção 1, será utilizado o método da classe Conta e, caso digite 3, será utilizado o método da classe ContaPoupanca.

Nesse exemplo, a mesma linha de código pode ter um comportamento diferente, dependendo das circunstâncias. Isso é polimorfismo!

Agora que entendemos o conceito de sobrescrita, vamos corrigir uma falha que cometemos ao definir nossa hierarquia de classes! A classe *ContaEspecial* tem um atributo *limite* que define um valor que o proprietário da conta poderia *sacar* mesmo não tendo saldo. O problema é que quando implementamos a classe *ContaEspecial* não reescrevemos o método sacar; logo, essa classe está utilizando o método da superclasse *Conta*. Assim, independentemente do valor do atributo limite da ContaEspecial, o saque não será efetuado caso não haja saldo suiciente, pois essa é a lógica implementada no método sacar da classe Conta. A Figura 41 exibe uma implementação para o método sacar na classe ContaEspecial que sobrescreve o método da superclasse e permite a realização do saque caso o valor a ser sacado seja menor ou igual a soma entre o saldo e o limite da conta.

```
@Override
public boolean sacar(double valor){
    if (valor <= this.limite + this.saldo) {
        this.saldo -= valor;
        return true;
    } else {
        return false;
    }
}
```

Figura 41: Método sacar sobrescrito na classe ContaEspecial

5.4 Sobrecarga

Métodos de mesmo nome podem ser declarados na mesma classe, contanto que tenham diferentes conjuntos de parâmetros (determinado pelo número, tipos e ordem dos parâmetros). Isso é chamado sobrecarga de método (DEITEL; DEITEL, 2010, p. 174).

Para que os métodos de mesmo nome possam ser distinguidos, eles devem possuir assinaturas diferentes. A assinatura (signature) de um método é composta pelo nome do método e por uma lista que indica os tipos de todos os seus argumentos. Assim, métodos com mesmo nome são considerados diferentes se recebem um diferente número de argumentos ou tipos diferentes de argumentos e têm, portanto, uma assinatura diferente.

Quando um método sobrecarregado é chamado, o compilador Java seleciona o método adequado examinando o número, os tipos e a ordem dos argumentos na chamada (DEITEL; DEITEL, 2010, p. 174).

Mesmo sem saber, nós já utilizamos o conceito de sobrecarga quando criamos os construtores da classe Conta. Ao criar a classe Conta nós definimos dois construtores, sendo que um deles recebendo três parâmetros e o outro recebendo dois parâmetros.

Para ilustrar melhor o conceito de sobrecarga, implementaremos na classe Conta um novo método *imprimirTipoConta* que receberá como parâmetro uma *String* e imprimirá na tela o tipo da conta seguido pela *String* recebida.
A Figura 42 exibe os dois métodos imprimirTipoConta da classe Conta.

```
public void imprimirTipoConta() {
    System.out.println("Conta Comum");
}
public void imprimirTipoConta(String s) {
    System.out.println("Conta Comum - String recebida:" + s);
}
```

Figura 42: Exemplo de sobrecarga

Na Figura 43 é apresentado um exemplo que visa ilustrar o comportamento dos objetos diante o uso de herança, sobrecarga e sobrescrita. A figura exibe tanto o código-fonte de uma classe que utiliza as classes Conta e ContaEspecial quanto a saída da execução desse programa (destacada em vermelho).

```
1   package banco;
2
3   public class UsaSobrecargaSobrescrita {
4
5       public static void main(String[] args) {
6           Conta c1 = new Conta(1, "Ze");
7           ContaEspecial c2 = new ContaEspecial(2, "João", 100);
8           c1.imprimirTipoConta();
9           c1.imprimirTipoConta("Teste Sobrecarga");
10          c2.imprimirTipoConta();
11          c2.imprimirTipoConta("Teste Sobrecarga em subclasse");
12      }
13  }
14
```

```
Saida - Aula4 (run)
run:
Conta Comum
Conta Comum - String recebida:Teste Sobrecarga
Conta Especial
Conta Comum - String recebida:Teste Sobrecarga em subclasse
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 43: Utilização de sobrecarga e sobrescrita

No exemplo da Figura 43, a variável c1 contém uma instância da classe Conta e c2, uma instância da classe ContaEspecial. Após a criação dos objetos, primeiro é feita uma chamada ao método imprimirTipoConta de c1 sem passar nenhum parâmetro, e depois outra chamada passando uma String. Em seguida foram feitas as mesmas chamadas a partir de c2. Note que como não criamos na classe ContaEspecial, um método imprimirTipoConta que receba uma String como argumento, a última chamada feita a partir de c2 foi atendida pelo método da classe Conta. O que aconteceu foi que, como c2 contém uma instância de ContaEspecial, o compilador procurou por um método imprimirTipoConta(String) na classe ContaEspecial. Como não encontrou, então o compilador fez uso do método existente na superclasse, já que ContaEspecial herda de Conta.