FIAP ON

JAVA FUNDAMENTOS

EXCEPTIONS + VÁRIOS OUTROS, **DESIGN PATTERS**

THIAGO YAMAMOTO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

6 EXCEPTIONS E CONCEITOS AVANÇADOS DE ORIENTAÇÃO A OBJETOS

6.1 Tratamento de exceções

Durante a execução de um programa é possível que algumas exceções ou erros aconteçam. Exceções e erros são problemas que podem ocorrer ao executar algum comando.

Abaixo estão listados alguns problemas mais comuns:

- Falha na aquisição de algum recurso:
    - Exemplo: abrir um arquivo, conectar com o banco de dados ou acessar um *web service*.
- Tentativa de realizar algo impossível:
    - Exemplo: divisão de um número por zero, acessar uma posição de um array que não existe.
- Outras condições inválidas:
    - Exemplo: evocar um método de um objeto que não foi instanciado, realizar um *cast* inválido.

Esses eventos não esperados geralmente interrompem o fluxo da execução do código. O tratamento de exceções permite verificar erros sem prejudicar o fluxo do programa, por exemplo, um programa que lê dois números para realizar uma divisão. O usuário pode inserir o número zero como divisor, assim, o programa lançará uma exceção, já que não é possível dividir um número por zero. Dessa forma, ao invés da exceção interromper o programa, podemos tratar esse possível cenário e pedir para o usuário inserir um outro número válido.

Em linhas gerais, o fluxo para o tratamento de exceções no Java ocorre em três passos:

- **Uma exceção é lançada:** um comando do código dispara uma condição inesperada de erro.
- **A exceção é capturada:** em algum ponto do código, podemos adicionar um comando para capturar a possível exceção.
- **O tratamento de erro é realizado:** após a captura da exceção, o tratamento de erro adequado é executado.

Realizando o tratamento das exceções, o programa consegue continuar a execução normalmente. Exceções não capturadas provocam a finalização do programa.

6.2 Classificação

Uma exceção é um objeto do tipo **Exception.** No polimorfismo, um objeto do tipo Exception pode ser qualquer instância de uma subclasse de Exception.

Dentro da plataforma Java, podemos classificar as exceções em:

- **Checked:** exceção que **deve** ser tratada ou relançada pelo desenvolvedor, geralmente herda da classe Exception.
- **Unchecked**: exceção que **pode** ser tratada ou relançada pelo programador. Essa exceção é gerada pelo código mal escrito, caso a exceção não for tratada, ela será automaticamente relançada. Geralmente esse tipo de exceção herda de RuntimeException.
- **Error**: erro crítico, utilizado pela JVM para indicar que existe um problema que não permite a execução do programa continuar.

O quadro abaixo apresenta um resumo das exceções no Java:

Classe	Objetivo	Exemplo
Error	Erro que não pode ser tratada na aplicação. Lançado pela JVM indicando que o programa não pode continuar a execução.	**OutOfMemoryError:** indica que não há memória suficiente na máquina para continuar a execução do programa.
Exception	Exceção que deve ser tratada pelo desenvolvedor.	**ArithmeticException:** indica que houve alguma operação aritmética inválida, por exemplo: divisão por zero.
RuntimeException	Exceção que pode ser tratada pelo desenvolvedor.	**NullpointerException:** indica que há uma tentativa de acessar algum método ou atributo de uma classe que não foi instanciada.

Quadro 6.1 – Resumo das exceções no Java
Fonte: Elaborado pelo autor (2017)

A figura abaixo apresenta a hierarquia de classes de exceções do Java. Além das exceções existentes na plataforma, podemos criar as nossas próprias exceções, sejam elas checked (filhas de exception) ou unchecked (filhas de RuntimeException):

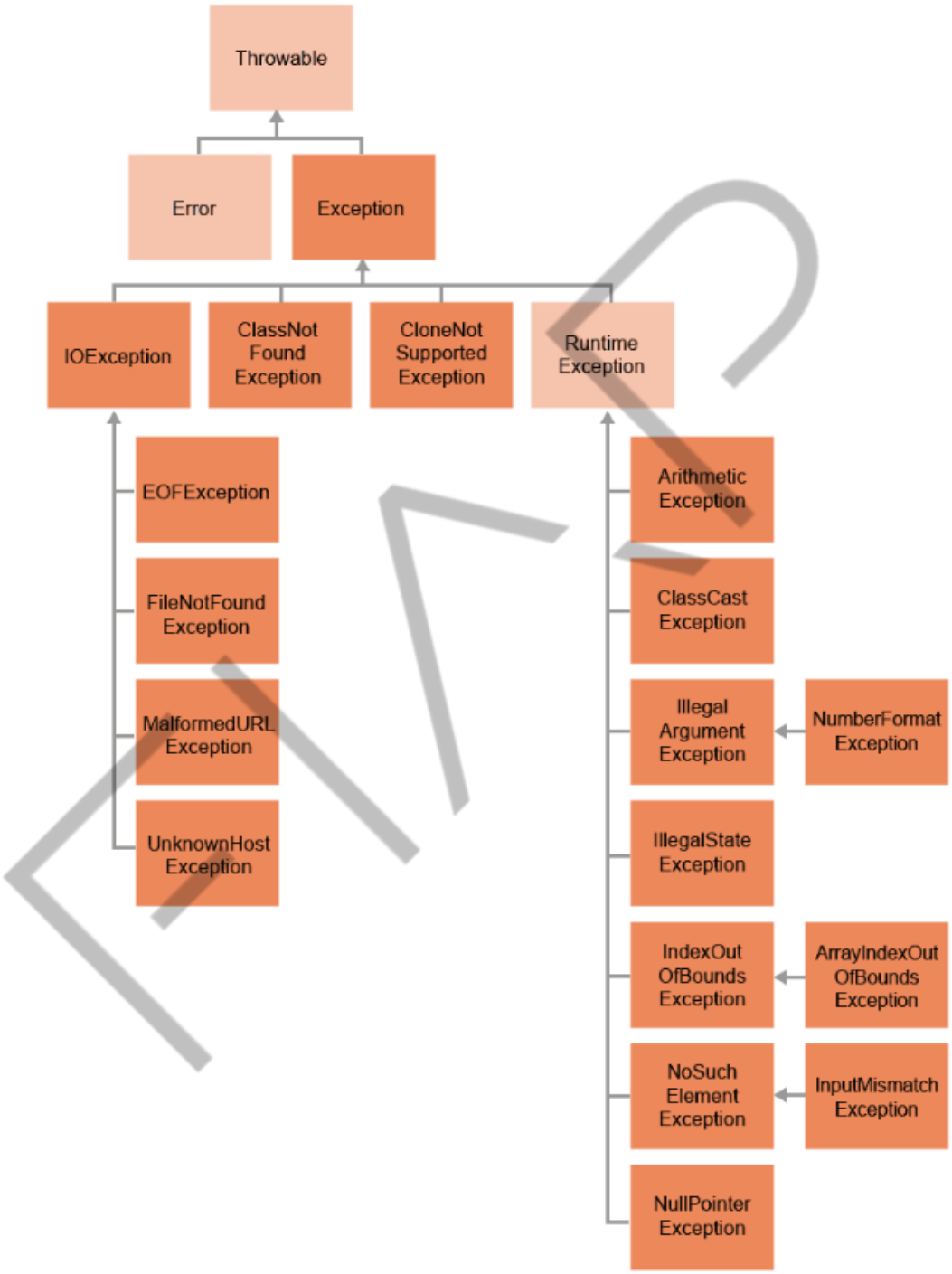

Figura 6.1 – The Hierarchy of Exception Classes
Fonte: Elaborado pelo autor (2017)

A classe Throwable possui duas subclasses: Exception e Error.

Exception é a classe base para as subclasses de exceções checked e unchecked, exceções que devem ou podem ser tratadas. Já a classe Error é a base para as classes de exceções que não podem ser tratadas.

Agora vamos descrever algumas das exceções mais comuns, existentes na plataforma Java:

- **ArithmeticException:** exceção *unchecked* que ocorre quando alguma operação aritmética é inválida, operação esta que não pode ser resolvida, como é o caso da divisão de um número por zero.
- **ArrayIndexOutOfBoundsException:** também é uma exceção do tipo *unchecked**,*** ou seja, o desenvolvedor não é obrigado a tratar a exceção. Esta exceção acontece quando tentamos acessar uma posição inválida de um array. Uma posição inválida é uma posição que não existe, como uma posição negativa ou um valor igual ou maior que o tamanho do vetor. Lembre-se que o índice do vetor sempre começa em zero, ou seja, não existe posição negativa e o último elemento de um array está posicionado no tamanho do vetor menos um.
- **NullpointerException:** uma exceção *unchecked*, mais conhecida e comum durante desenvolvimento. Ocorre na tentativa de acessar um objeto que ainda não foi instanciado. Por exemplo, quando tentamos acessar o método *size()* de um ArrayList que ainda não foi instanciado.
- **FileNotFoundException:** uma exceção *checked**,*** que precisamos tratar quando tentamos acessar um arquivo que não foi encontrado.
- **NumberFormatException:** exceção que ocorre quando tentamos transformar uma string inválida em algum tipo numérico. Esta exceção não precisa ser tratada. (*unchecked)*

6.3 Captura e Tratamento de Exceções

Para tratar as exceções (*checked ou unchecked)* em tempo de execução, elas devem ser capturadas e tratadas. O java possui duas estruturas importantes para o tratamento de exceções:

- try-catch
- try-catch-finally

Estas duas estruturas têm a finalidade de separar o código que executa as tarefas desejadas, das rotinas de tratamento das exceções:

```
try{
      //Código
}catch(Exceção){
      //Tratamento da exceção
}
```

O bloco **try** possui o código que pode gerar uma exceção, ou seja, este trecho de código será monitorado pela JVM. Se um erro for gerado, o fluxo da execução é desviado para o bloco **catch**, para o tratamento do erro. O uso do **try** indica que o código está tentando realizar algo “perigoso”, passível de erro.

O bloco **catch** só é executado se uma exceção for gerada. Caso nenhuma exceção seja lançada, a execução pula o bloco catch e continua normalmente. Se uma exceção for lançada, o bloco **try** é finalizado e o fluxo de execução procura por um bloco **catch** adequado para tratar a exceção, depois de executar o bloco **catch**, a execução continua na primeira instrução após o último bloco **catch**.

Podemos adicionar vários blocos **catch** para capturar diferentes tipos de exceções:

```
try{
      //Código
      }catch(Exceção 1){
            //Tratamento da exceção 1
      }catch(Exceção 2){
            //Tratamento da exceção 2
      }catch(Exceção 3){
            //Tratamento da exceção 3
}
```

Dessa forma, somente o primeiro **catch** que se encaixar com o tipo da exceção lançada será executado. Os **catchs** são testados de cima para baixo, um por um, por isso, as exceções mais específicas devem ser colocadas nos primeiros **catchs**, sempre obedecendo à ordem: das exceções mais específicas para as mais genéricas.

A exceção **Exception** é a mais genérica possível, pois todas as exceções (*checked ou unchecked)* são filhas dela. Portanto, a captura dessa exceção deve ser colocada no último catch, pois ela captura qualquer tipo de exceção que for lançada.

Se nenhum **catch** conseguir capturar a exceção lançada, ela não será tratada, como se não existisse o bloco **try-catch.**

Vamos desenvolver um exemplo: Ler dois números para dividir um pelo outro:

```
            Scanner sc = new Scanner(System.in);

            //Lê os dois números
            int numero1 = sc.nextInt();
            int numero2 = sc.nextInt();

            //Realiza a divisão
            int divisao = numero1/numero2;

            //Exibe o resultado
            System.out.println("O resultado é: " + divisao);

        sc.close();
```

O código é bem simples, porém pode lançar uma exceção, caso o segundo número informado pelo usuário for zero. Caso isso aconteça, o erro gerado será:

```
2
0
Exception in thread "main" java.lang.ArithmeticException: / by
zero
        at br.com.fiap.tds.View.main(View.java:16)
```

A exceção **ArithmeticException** foi lançada, pois não é possível realizar uma divisão por zero. Podemos perceber que essa exceção é *unchecked,* já que não fomos obrigados a tratá-la.

Vamos realizar uma pequena alteração no código, para realizar o tratamento da exceção:

```
        Scanner sc = new Scanner(System.in);

            // Lê os dois números
            int numero1 = sc.nextInt();
            int numero2 = sc.nextInt();

            try {
                // Realiza a divisão
                int divisao = numero1 / numero2;
                // Exibe o resultado
                System.out.println("O resultado é: " + divisao);
            } catch (ArithmeticException e) {
```

```
                System.err.println("Erro ao dividir!");
            }
            sc.close();
```

Dessa vez, adicionamos um bloco **try-catch** para capturar a exceção. Caso o usuário insira um divisor igual a zero, a divisão vai gerar a ArithmeticException. Dessa forma, o fluxo da execução será desviado para o bloco catch, (o código que imprime o resultado não será executado) onde será exibida a mensagem de erro.

O resultado da execução do programa, com o divisor igual a zero:

```
2
0
Erro ao dividir!
```

Se o programa rodar sem lançar exceção, o resultado será exibido e o bloco catch não será executado.

Dentro do bloco catch podemos recuperar a exceção gerada, através do parâmetro. No exemplo acima, a exceção é recuperada no parâmetro com o nome "e".

A classe **Throwable** possui alguns métodos que podem exibir informações dos erros gerados. Portanto, por herança, a exceção ArithmeticException também possuí esses métodos:

- **printStackTrace() –** imprime a pilha de erro encontrada na exceção, nesta pilha, podemos verificar o número da linha e classe onde a exceção foi gerada;
- **getMessage() –** retorna uma mensagem contento a lista de erros armazenadas em uma exceção.

No exemplo abaixo, vamos imprimir o *StackTrace* e a mensagem de erro da exceção:

```
int[] array = new int[2];

        try {
            //Tenta acessar uma posição inexistente do vetor
            array[2] = 10;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Mensagem de erro: " +
e.getMessage());
```

```
            e.printStackTrace();
    }
```

Dessa vez, a exceção será gerada quando tentamos acessar uma posição inválida de um array: **ArrayIndexOutOfBoundsException**.

No bloco catch, utilizamos o método getMessage() e printStackTrace(), o resultado será:

```
Mensagem de erro: 2
java.lang.ArrayIndexOutOfBoundsException: 2
    at br.com.fiap.tds.View.main(View.java:10)
```

A mensagem possui o valor do índice inválido que tentamos acessar no array. O printStackTrace imprime o rastro da pilha e exibe a exceção, a classe e a linha que gerou a exceção: **View.java:10.** Classe **View**, linha **10**.

Observe que utilizamos o **System.err** para exibir a mensagem de erro. Essa classe é utilizada para exibir os erros em um programa Java. A saída de mensagem possui a cor vermelha na visualização de mensagem. Podemos enviar o fluxo de saída para um arquivo de log, enquanto que o fluxo de saída padrão de um **System.out** é a tela da aplicação.

Além do bloco **try-catch,** podemos utilizar o bloco **try-catch-finally** para tratar as exceções. O terceiro bloco **finally** é opcional e é utilizado quando sempre precisamos executar um código, independentemente se aconteceu ou não uma exceção:

- **Se uma exceção for lançada:** o fluxo de execução passa do bloco **try** para o bloco **catch** adequado e após a sua execução, o bloco **finally** será processado.
- **Se não for gerado uma exceção:** após executar todo o bloco **try**, o bloco **finally** será processado.

O bloco **finally** não é obrigatório e quando usado deve sempre ser colocado no final do último bloco **catch.**

Formato geral:

```
try {
            //Fluxo normal que pode gerar uma exceção
     } catch (Exception e) {
```

```
            //Fluxo alternativo, para tratamento da exceção
        } finally{
            //Fluxo normal,que sempre será executado
        }
```

Um exemplo clássico da utilização do bloco **finally** é o fechamento da conexão com o banco de dados. Sempre devemos fechar a conexão, independentemente da concretização ou não da operação no banco.

Voltando ao primeiro exemplo, vamos tratar as exceções de divisão por zero e entrada de dados inválida, quando o usuário digita um número inválido, como uma letra. Também vamos adicionar um bloco *finally* para exibir uma mensagem para o usuário e fechar o objeto *scanner*.

```
        Scanner sc = new Scanner(System.in);
        try {
            // Lê os dois números
            int numero1 = sc.nextInt();
            int numero2 = sc.nextInt();
            // Realiza a divisão
            int divisao = numero1 / numero2;
            // Exibe o resultado
            System.out.println("O resultado é: " + divisao);
        } catch (ArithmeticException e) {
            System.err.println("Erro ao dividir!");
        } catch (InputMismatchException e) {
            System.err.println("Erro de entrada de dados!");
        } finally{
            System.out.println("Finalizando a execução do
programa!");
            sc.close();
    }
```

Dessa forma, temos três possíveis fluxos de execução:

- Sem gerar exceção, a saída será:

```
2
2
O resultado é: 1
Finalizando a execução do programa!
```

- Exceção causada pela divisão por zero, a saída será:

```
2
0
Erro ao dividir!Finalizando a execução do programa!
```

- Exceção causada pelo valor inserido inválido, a saída será:

```
a
Erro de entrada de dados!Finalizando a execução do
programa!
```

Observe que todos os fluxos sempre executam o bloco finally.

6.4 Propagação de exceções – Throws

Um método pode optar por não tratar a exceção e simplesmente propagá-la, ou melhor, delegar para o método que a chamou. Dessa forma, podemos notificar o método que invocou outro método, que alguma exceção ocorreu. Por exemplo, vamos criar uma classe chamada Calculadora que será responsável por implementar as operações aritméticas, entre eles a divisão:

```
public class Calculadora {

        public int dividir(int n1, int n2){
            return n1/n2;
        }

}
```

Esse método pode lançar uma exceção, caso o valor de n2 for zero. Podemos utilizar o try-catch para tratar a exceção, porém quem chamar o método **dividir** não saberá se a operação ocorreu de forma correta ou se aconteceu algum erro:

```
public int dividir(int n1, int n2) {
        try {
            return n1 / n2;
        } catch (ArithmeticException e) {
            e.printStackTrace();
        }
        return 0;
}
```

Dessa forma, a melhor maneira de tratar a exceção é não tratar. Neste caso, devemos somente propagar a exceção, notificando assim que algum problema aconteceu na execução. Para isso, devemos adicionar na assinatura do método o **throws,** junto da exceção que queremos propagar:

```
public int dividir(int n1, int n2) throws Exception{
        return n1 / n2;
}
```

Um método pode propagar mais de um tipo de exceção, para isso, basta adicionar as exceções separadas por vírgula:

```
public void gravarArquivo(String valor) throws
SecurityException, FileNotFoundException, IOException{
        //Código...
}
```

Portanto, a cláusula **throws** declara exceções que podem ser lançadas em determinados métodos. Isso é uma vantagem para os desenvolvedores, pois deixamos de modo explícito os eventuais erros que podem ocorrer na chamada do método, permitindo que o tratamento adequado para o erro seja implementado.

Podemos também lançar uma nova exceção no nosso método. Para isso, basta utilizar o comando **throw:**

```
public void depositar(double valor){
        if (valor < 0){
                throw new IllegalArgumentException();
        }
        saldo = saldo + valor;
}
```

No exemplo acima, estamos validando se o valor depositado é maior do que zero. Se for, o valor é adicionado ao saldo, caso contrário, uma exceção do tipo IllegalArgumentException será lançada. Isso indica que o valor passado como parâmetro para o método é inválido. Repare que não foi preciso adicionar o **throws** na assinatura do método, pois essa exceção é *unchecked.*

Caso a exceção seja *checked,* é necessário declará-la na assinatura do método:

```
public void sacar(double valor) throws Exception{
        if (valor > saldo){
                throw new Exception("Saldo insuficiente");
```

```
            }
            saldo = saldo - valor;
}
```

O método acima valida se o valor a ser retirado é maior do que o valor do saldo. Caso o valor do saldo seja insuficiente, uma exception será lançada. Para isso, foi necessário adicionar o **throws** na assinatura do método. Dessa forma, quem chamar o método sacar deve tratar a exceção ou lançá-la novamente:

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

            // Cria uma nova instância de Conta
            Conta c = new Conta();
            try {
                  // Saca
                  c.sacar(100);
            } catch (Exception e) {
                  e.printStackTrace();
            }
            // Deposita
            c.depositar(200);

}
```

Após criar um novo objeto conta, chamamos o método sacar, como ela lança uma exceção *checked***,** precisamos tratá-la ou lançá-la, neste caso, optamos por tratar a exceção com o bloco **try-catch**. Já o método depositar lança uma exceção *unchecked*, e dessa forma, não fomos obrigados a tratá-la. Porém, se uma exceção ocorrer, ela será relançada automaticamente pelo método **main** que irá imprimir o erro no console.

6.5 Criação de exceções

É possível também criar as nossas próprias classes de exceções. Observe o exemplo acima, o método **sacar** lança a exceção mais genérica possível (Exception). Para melhorar o uso das exceções e não utilizar uma exceção que serve para tudo, podemos criar uma exceção específica para o erro que pode acontecer.

Para criar uma exceção, basta criar uma classe que herde de Exception (*checked)* ou RuntimeException (*unchecked)*.

Vamos criar as nossas exceções, uma para identificar que o valor de saque é inválido e outra para dizer que o saldo da conta é insuficiente.

A primeira exception será *unchecked,* ou seja, ela vai descender de RuntimeException. Por padrão, as exceções no Java terminam com Exception:

```
public class ValorInvalidoException extends RuntimeException {

}
```

Para utilizá-la, vamos modificar o método de sacar, para que ela lance a nossa exceção customizada:

```
public void depositar(double valor){
            if (valor < 0){
                  throw new ValorInvalidoException();
            }
            saldo = saldo + valor;
}
```

Agora vamos criar uma exceção *checked* que identifica que o saldo é insuficiente:

```
public class SaldoInsuficienteException extends Exception{

}
```

Para utilizá-la, podemos relançá-la, não se esquecendo de modificar o **throws**, na assinatura do método:

```
 public       void       sacar(double       valor)       throws
SaldoInsuficienteException{
            if (valor > saldo){
                  throw new SaldoInsuficienteException();
            }
            saldo = saldo - valor;
 }
```

Nos exemplos acima, criamos exceções *checked* e *unchecked* para ilustrar nossos exemplos. Mas nada impediria de criar somente um tipo de exceção para a nossa classe Conta. Procure sempre utilizar as exceções customizadas com informações do motivo do erro.

6.6 Acesso a arquivos

O armazenamento de dados em variáveis, arrays, coleções ou qualquer outra estrutura de dados em memória é temporário. Todas as informações armazenadas em memória são perdidas quando o programa termina. Os arquivos são utilizados para persistência de dados em longo prazo, mesmo após o término da execução do programa.

Para armazenar grande quantidade de informações, como os clientes, produtos ou vendas de uma empresa, os lugares mais adequados são os bancos de dados, assunto que será abordado no próximo capítulo.

Os arquivos são muito úteis para armazenar as configurações do programa, ao invés de colocar as configurações diretamente no código fonte. A grande vantagem disso é a possibilidade de alterar o arquivo de configurações sem a necessidade de recompilar e empacotar todo o programa novamente. Algo impossível, se a configuração estivesse diretamente no código fonte.

Outra possibilidade de utilização de arquivos é a integração de sistemas diferentes, algo muito utilizado. Um sistema pode realizar um processamento de informações e gerar um arquivo final. Um outro sistema pode ler esse arquivo para apresentar as informações, gerando assim a integração. Por exemplo, para calcular a fatura de telefone, uma empresa pode utilizar um sistema de alto processamento que gera um arquivo final com a fatura consolidada. Outro sistema lê esse arquivo e exibe as informações na internet para o usuário final.

A plataforma Java possui uma facilidade na leitura e gravação de arquivos, pois como ela é independente de plataforma, não precisamos nos preocupar com o tipo de sistema operacional que o programa será executado. E claro, com uma linguagem orientada a objetos, vamos utilizar objetos que realizam todo o trabalho para nós.

Todas as entradas e saídas em Java são definidas em termos de fluxos, ou *streams,* que são sequências ordenadas de dados que possui uma fonte, no caso de *streams* de entrada, ou um destino, no caso de *streams* de saída.

Em outras palavras, uma *stream* é uma conexão para uma fonte de dados ou para um destino de dados. A plataforma Java trata cada arquivo como uma *stream.*

A figura abaixo exibe os dois tipos de *streams*, dependendo do tipo de operação que será realizada:

- ***Output stream:*** para gravar em um destino.
- ***Input stream:*** para ler de uma fonte.

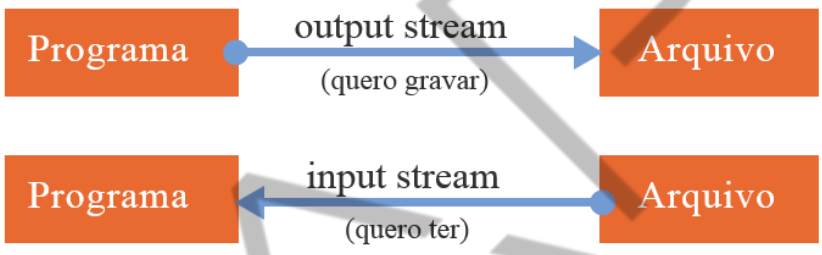

Figura 6.2 – Streams
Fonte: Elaborado pelo autor (2017)

Arquivos baseados em um fluxo de caracteres são arquivos textos:

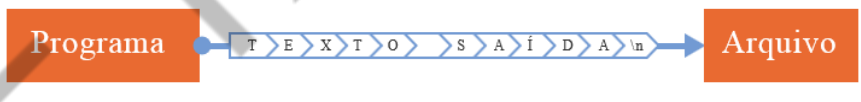

Figura 6.3 – Arquivos textos
Fonte: Elaborado pelo autor (2017)

Um arquivo texto pode ser lido e entendido facilmente pelas pessoas e também de ser manipulado na plataforma Java. Podem ser utilizados facilmente por diversos programas, pois é fácil saber a disposição dos registros e campos contidos nestes arquivos. Geralmente, os registros nestes arquivos são representados pelas linhas, enquanto os campos são representados por colunas ou valores separados por vírgula:

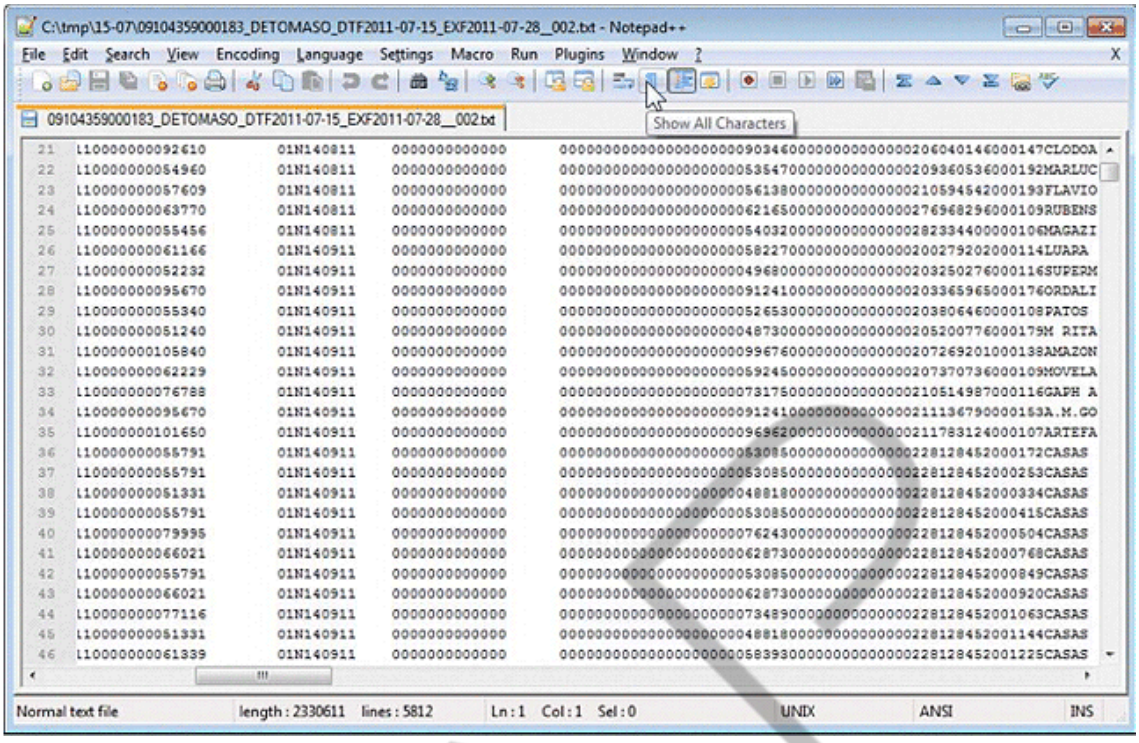

Figura 6.4 – Exemplo de arquivo texto
Fonte: Elaborado pelo autor (2017)

Basicamente, para gravar dados em um arquivo texto, é preciso executar três passos:

- Abrir o arquivo.
- Gravar os dados.
- Fechar o arquivo.

As classes para manipular os arquivos ficam dentro do pacote **java.io.** Duas classes que podemos utilizar para a escrita em arquivo são: java.io.FileWriter e java.io.PrintWriter.

A classe FileWriter é filha da classe OutputStreamWriter. Vamos utilizar essa classe para abrir o arquivo para a escrita. Depois, vamos utilizar a classe PrintWriter para escrever no arquivo. Para criar um objeto do tipo PrintWriter é preciso passar o arquivo, ou seja o objeto FileWriter:

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

		try {
			//Abre o arquivo
			FileWriter stream = new
FileWriter("arquivo.txt");
```

```
                PrintWriter print = new PrintWriter(stream);

                //Escreve no arquivo
                print.println("Teste");
                print.println("Escrevendo no arquivo");

                print.close();
                //Fecha o arquivo
                stream.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

    }
```

O exemplo acima apresenta o método main que abre, escreve e fecha um arquivo chamado “arquivo.txt”. Esse arquivo será aberto se existir, ou criado, caso não exista. Observe que na instanciação do FileWriter é passado o caminho e nome do arquivo. Como não especificamos nenhum caminho, o arquivo estará na pasta raiz do programa:

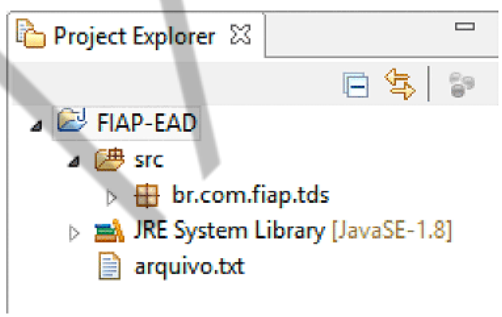

Figura 6.5 – Exemplo de pasta raiz do programa
Fonte: Elaborado pelo autor (2017)

Podemos também especificar o caminho completo do arquivo, por exemplo:

```
FileWriter stream = new FileWriter("C://arquivo.txt");
```

Após criar o objeto FileWriter, foi instanciado o objeto PrintWriter, passando com parâmetro o objeto que encapsula o nosso arquivo:

```
PrintWriter print = new PrintWriter(stream);
```

Com o objeto PrintWriter é possível utilizar os métodos print ou println para escrever no arquivo:

```
print.println("Teste");
      print.println("Escrevendo no arquivo");
```

Para finalizar chamamos o método **close** do objeto PrintWriter e FileWriter, para fechar o arquivo texto:

```
print.close();
stream.close();
```

Observe que foi preciso tratar a exceção **IOException,** pois podemos ter problemas na hora de abrir ou manipular o arquivo.

O resultado da execução será o arquivo texto com as informações gravadas:

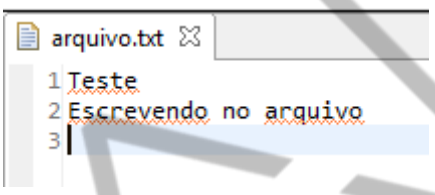

Figura 6.6 – O resultado da execução
Fonte: Elaborado pelo autor (2017)

Para ler o arquivo, devemos realizar três passos também:

- Abrir o arquivo.
- Utilizar o arquivo (ler os dados).
- Fechar o arquivo.

As classes que podemos utilizar para ler os dados de um arquivo são: java.io.FileReader e java.io.BufferedReader. A classe FileReader descende da classe InputStreamReader.

O exemplo abaixo abre o arquivo criado no exemplo anterior, o “arquivo.txt”, lê cada linha do arquivo e exibe no console.

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

            try {
                  //Abre o arquivo
                  FileReader stream = new
FileReader("arquivo.txt");
```

```
                    BufferedReader reader = new
BufferedReader(stream);

                    //Lê uma linha do arquivo
                    String linha = reader.readLine();
                    while (linha != null){
                          System.out.println(linha);
                          //Lê a próxima linha do arquivo
                          linha = reader.readLine();
                    }

                    reader.close();
                    //Fecha o arquivo
                    stream.close();
              } catch (IOException e) {
                    e.printStackTrace();
              }

        }
```

Assim como no exemplo de escrita de arquivo, precisamos abrir o arquivo, mas dessa vez utilizarmos a classe FileReader ao invés do FileWriter. Também foi necessário passar o nome do arquivo que queremos abrir como parâmetro. A mesma regra do caminho se aplica, caso o arquivo esteja em outro diretório, devemos especificá-lo:

```
FileReader stream = new FileReader("C://arquivo.txt");
```

Se o arquivo não for encontrado, será gerada a exceção **FileNotFoundException.**

Após criar o objeto FileReader, foi instanciado o objeto **BufferedReader** para realizar a leitura do arquivo. Para criar esse objeto, passamos como parâmetro o objeto FileReader criado anteriormente:

```
BufferedReader reader = new BufferedReader(stream);
```

A classe *BufferedReader* possui o método **readLine**(), ele lê uma linha do arquivo e retorna uma *String* com o valor lido ou ***null,*** caso o arquivo não possua mais linhas para serem lidas.

Dessa forma, utilizamos esse método para ler uma linha e depois criamos um laço de repetição, que vai ser executado até que a String recuperada for nula, ou seja, até que todas as linhas do arquivo forem lidas.

```
//Lê uma linha do arquivo
String linha = reader.readLine();
while (linha != null){
      System.out.println(linha);
      //Lê a próxima linha do arquivo
      linha = reader.readLine();
}
```

Para finalizar, é preciso fechar o FileReader e o BufferedReader:

```
reader.close();
      //Fecha o arquivo
      stream.close();
```

A plataforma Java oferece várias Classes para a leitura/escrita de arquivos texto. Caso implementação é destinada a uma aplicação específica, essas classes ficam dentro do pacote *java.io:*

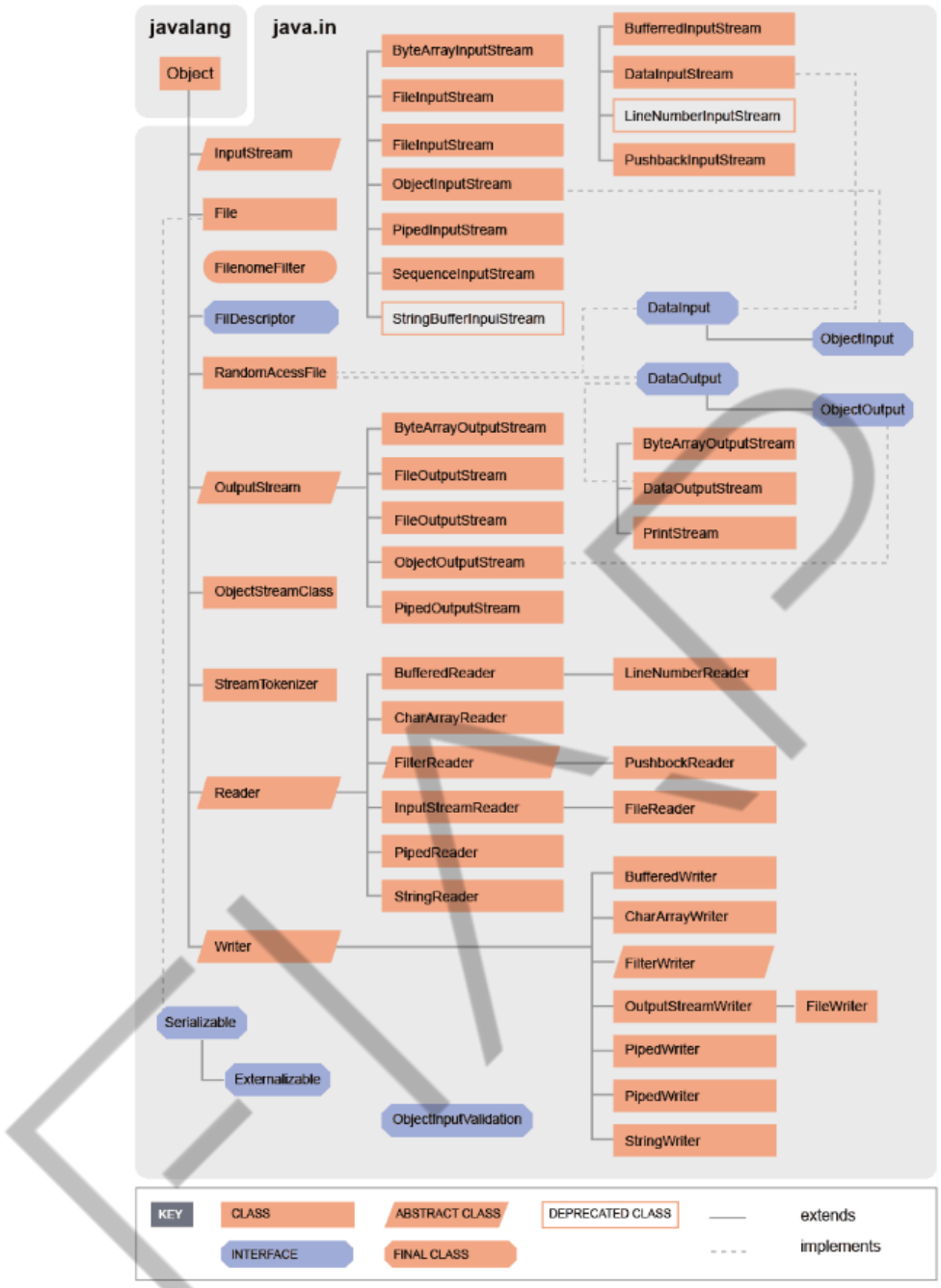

Figura 6.7 – Classes
Fonte: Elaborado pelo autor (2017)

As principais classes desse pacote são:

- Classes para entrada ou saída baseada em bytes:
  - **FileInputStream:** para entrada baseada em bytes de um arquivo.
  - **FileOutputStream:** para saída baseada em bytes para um arquivo.

- **RandomAccessFile:** para entrada e saída baseada em bytes de e para um arquivo.

- Classes para entrada e saída baseada em caracteres:
  - **FileReader:** para entrada baseada em caracteres de um arquivo.
  - **FileWriter:** para saída baseada em caracteres para um arquivo

Outra classe importante da API de IO do Java é o **File**. Essa classe representa um arquivo ou um diretório no sistema de arquivo nativo. Ela permite obter informações sobre o arquivo ou diretório, e não sobre o seu conteúdo.

Ao contrário das outras classes que estudamos, esta classe não abre o arquivo ou permite o processamento de seu conteúdo. A sua função é gerenciar o arquivo ou diretório, o seu caminho (*path)*, verificar permissões, a existência, criar, renomear etc.

Um objeto da classe File apenas representa o arquivo ou diretório, isto não quer dizer que o arquivo ou diretório exista de fato.

O quadro abaixo apresenta os principais métodos da classe File:

Método	Descrição
exists	verifica se um arquivo ou diretório existe
isDirectory	verifica se é um diretório
isFile	verifica se é um arquivo
canRead	verifica se pode ser lido
canWrite	verifica se pode ser gravado
mkdir	cria um diretório
renameTo	renomeia um arquivo ou diretório
length	retorna o tamanho do arquivo
delete	deleta um arquivo ou diretório
getPath	retorna o caminho

Quadro 6.2 – Os principais métodos da classe File
Fonte: Elaborado pelo autor (2017)

O exemplo abaixo cria um objeto do tipo File, passando o nome do arquivo, é possível também passar o nome do arquivo e o seu caminho. O código verifica se o arquivo existe. Se existir, ele recupera algumas informações a respeito do arquivo e exibe no console. Caso não exista, o programa tenta criar o arquivo:

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

```
            File arquivo = new File("arquivo.txt");

            // Verifica se o arquivo existe
            if (arquivo.exists()) {
                  System.out.println("O arquivo existe!"+
                              "\nPode ser lido: " + arquivo.canRead() +
                              "\nPode ser gravado: " + arquivo.canWrite()
+
                              "\nTamanho: " + arquivo.length() +
                              "\nCaminho: " + arquivo.getPath());
            } else {
                  // Cria o arquivo
                  try {
                        if (arquivo.createNewFile())
                              System.out.println("Arquivo criado!");
                        else
                         System.out.println("Arquivo não criado!");
                  } catch (IOException e) {
                        e.printStackTrace();
                  }
            }
      }
```

Se o arquivo for criado com sucesso, o método **createNewFile()** retorna o valor *true,* caso contrário *false.*

Podemos também utilizar o objeto File para verificar se um diretório existe ou para criá-lo. O código abaixo faz exatamente isso:

```
      public static void main(String[] args) {
            File diretorio = new File("fiap");

            if (diretorio.exists()){
                  System.out.println("Diretório existe!");
            }else{
                  if (diretorio.mkdir())
                        System.out.println("Diretório criado!");
                  else
                        System.out.println("Diretório não criado.");
            }
      }
```

O objeto File foi instanciado com o valor “fiap”, que é o nome do diretório. Primeiro verificamos se o diretório existe, caso não exista, utilizamos o método **mkdir()** para criar o diretório. Esse método retorna um valor booleano, verdadeiro se o diretório foi criado ou false, caso contrário.

Após criar um diretório, podemos criar um arquivo dentro dele, conforme o código abaixo:

```
	public static void main(String[] args) {
		File diretorio = new File("fiap");

		if (diretorio.exists()){
			System.out.println("Diretório existe!");
		}else{
			if (diretorio.mkdir())
				System.out.println("Diretório criado!");
			else
				System.out.println("Diretório não
criado.");
		}

		File arquivo = new File(diretorio,"file.txt");
		try {
			if (arquivo.createNewFile())
				System.out.println("Arquivo criado!");
			else
				System.out.println("Arquivo não criado!");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
```

Observe que primeiro verificamos se o diretório existe e se caso necessário, criamos. Depois instanciamos uma classe File passando o nome do arquivo e o objeto File que representa o diretório:

```
File arquivo = new File(diretorio,"file.txt");
```

Com esse novo objeto File, tentamos criar o arquivo dentro do diretório.

A classe **File** pode ser utilizada em conjunto com as outras classes de manipulação do conteúdo do arquivo, vistos acima. Para isso, basta utilizar o objeto **File** no momento de criar um **FileWriter** ou **FileReader**:

```
try {
            //Abre o arquivo para escrita
            FileWriter writer = new FileWriter(arquivo);
            //Abre o arquivo para leitura
            FileReader reader = new FileReader(arquivo);
            //Código...
      } catch (IOException e) {
            e.printStackTrace();
      }
```

Depois, basta instanciar a classe PrintWriter para escrever no arquivo ou BufferedReader para ler.

6.7 Polimorfismo

A palavra polimorfismo significa: "*qualidade ou estado de ser capaz de assumir diferentes formas"* dicionário Houaiss. Na programação orientada a objetos, polimorfismo significa ter múltiplos comportamentos.

Um método polimórfico resulta em diferentes ações dependendo do objeto que está sendo referenciado. A capacidade polimórfica decorre diretamente da herança, pois permite que uma variável de referência e o objeto sejam diferentes, ou seja, podemos definir uma variável do tipo **Conta** e armazenar um objeto de **ContaCorrente,** se este herdar da classe **Conta**. Portanto, o tipo da variável de referência pode ser uma superclasse para o tipo do objeto real. Qualquer objeto de uma classe que herde o tipo declarado da variável pode ser atribuído a ela.

Com o polimorfismo, é possível escrever um código que não tenha que ser alterado quando novos tipos de subclasse forem introduzidos no sistema. Porém, a utilização mais importante do polimorfismo se dá quando dois objetos, sendo um da superclasse e outro da subclasse, executam ações diferentes quando o mesmo método é invocado. Isso é possível através da sobrescrita de métodos, quando a subclasse sobrescreve o método implementado na superclasse. Já vimos isso nos primeiros capítulos de orientação a objetos.

Observe as classes abaixo:

```
 public class Conta {

protected double saldo;
```

```
public void sacar(double valor) throws SaldoInsuficienteException{
            if (valor > saldo){
                  throw new SaldoInsuficienteException();
            }
            saldo = saldo - valor;
}

 }

 public class ContaCorrente extends Conta{

      private double limite;

      @Override
      public void sacar(double valor) throws SaldoInsuficienteException {
            if (valor > saldo + limite){
                  throw new SaldoInsuficienteException();
            }
            saldo = saldo - valor;
      }

 }
```

Neste exemplo, a classe **Conta** possui um atributo que armazena o saldo e um método que realiza saque. Observe que o método pode lançar uma exceção, caso o valor a ser sacado é maior do que o saldo.

A classe **ContaCorrente** herda da classe conta, ou seja, a superclasse é a *Conta* e a subclasse *ContaCorrente.* A subclasse adiciona um outro atributo que é o limite da conta. O método *sacar* foi sobrescrito, agora a exceção é lançada caso o valor a ser sacado seja maior que o saldo mais o limite disponível.

No exemplo abaixo, definimos uma variável de referência do tipo *Conta* e atribuímos um objeto do tipo *ContaCorrente*:

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

            Conta cc = new ContaCorrente();
            try {
                  cc.sacar(20);
            } catch (SaldoInsuficienteException e) {
                  e.printStackTrace();
            }

}
```

Depois foi chamado o método **sacar.** Qual método será executado? O método definido na classe Conta ou da classe ContaCorrente?

O método do objeto armazenado na variável será executado. Ou seja, o método definido na *ContaCorrente*. Sempre a execução acontece com o objeto armazenado e não com o tipo da variável.

6.8 Classe abstrata

No nosso exemplo, a classe Conta é uma conta genérica, que serve como base para os outros tipos de conta como a conta corrente, conta poupança, conta investimento etc. Por isso, não faz sentido instanciar uma classe Conta, pois não existe uma “Conta Genérica” no nosso sistema. O que podemos instanciar são as subclasses da classe conta. É ai que entra a classe Abstrata.

A palavra abstrata significa que possui alto grau de generalização (dicionário Houaiss). Uma classe abstrata possui algumas características como:

- Não pode ser instanciada.
- **Pode** possuir métodos abstratos.

Logo, vamos discutir sobre métodos abstratos, porém, observe o destaque da palavra “**pode”,** isso significa que a classe pode conter esse tipo de método e não que **deve**.

A primeira característica é que uma classe abstrata não pode ser instanciada, ou seja, não podemos utilizar o operador **new.** Dessa forma, nunca vamos ter uma instancia de uma classe abstrata dentro da nossa aplicação.

O propósito de uma classe abstrata é atuar como uma superclasse. É uma classe que existe para ser herdada. Dessa forma, ela será a base para as outras classes que serão desenvolvidas.

Então a nossa classe **Conta** é perfeita para ser abstrata. Ela será a base para todas as outras contas em nosso sistema.

Como visto no exemplo acima, podemos definir uma variável com o tipo da classe abstrata, porém, devemos atribuir um objeto de uma subclasse da mesma, a fim de utilizar o polimorfismo.

Uma classe que não é abstrata é chamada de classe Concreta. Ao projetar uma hierarquia de herança, devemos definir quais classes serão concretas e abstratas.

Uma classe abstrata pode conter **métodos abstratos**. O método abstrato não possui implementação, ela define somente a assinatura do método. A sua subclasse, obrigatoriamente, precisará implementar o método, caso esta seja concreta.

Em uma classe concreta não é permitido definir métodos abstratos. Se isso acontecer, obrigatoriamente a classe deverá ser abstrata também. E isso faz sentido, pois se fosse possível ter um método sem implementação em uma classe concreta, o que aconteceria se você instanciar a classe e chamar este método? Difícil dizer. Por essa razão que a classe também deve ser abstrata, para não ser possível instanciá-la.

É possível ter uma herança com várias classes abstratas, porém a primeira classe concreta da hierarquia será obrigada a implementar todos os métodos abstratos definidos pelas suas superclasses.

Implementar o método abstrato é igual a sobrescrever um método, devemos definir um método com o mesmo nome e parâmetros (assinatura) na subclasse e implementá-la.

A figura abaixo representa uma classe abstrata no diagrama de classes:

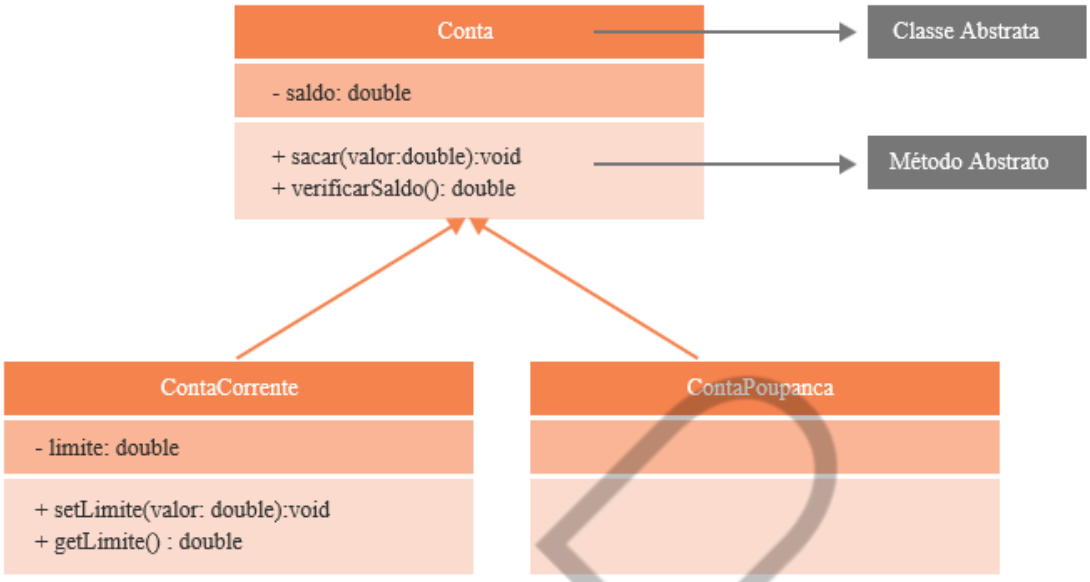

Figura 6.8 – Classe abstrata no diagrama de classes
Fonte: Elaborado pelo autor (2017)

Observe que o nome da classe abstrata e o método abstrato ficam em itálico. As classes concretas e métodos implementados são exibidos sem nenhuma formatação diferente.

Para definir uma classe ou método abstrato basta adicionar o modificador ***abstract*.**

Observe o exemplo abaixo, vamos definir a classe Conta como abstrata e adicionar o método *depositar* como abstrato:

```
public abstract class Conta {
    protected double saldo;

    public void sacar(double valor) throws SaldoInsuficienteException{
        if (valor > saldo){
            throw new SaldoInsuficienteException();
        }
        saldo = saldo - valor;
    }

public abstract double verificarSaldo();

 }
```

O método *verificarSaldo* é abstrato, portanto não possui implementação. Dessa forma, a assinatura do método termina com ponto e vírgula (;).

Agora a subclasse *ContaCorrente* está com erro de compilação devido à adição do método abstrato na classe Conta. Para arrumar o problema, devemos implementar o método *verificarSaldo* na classe *ContaCorrente*:

```
public class ContaCorrente extends Conta{

	private double limite;

	@Override
	public void sacar(double valor) throws SaldoInsuficienteException {
		if (valor > saldo + limite){
			throw new SaldoInsuficienteException();
		}
		saldo = saldo - valor;
	}

	@Override
	public double verificarSaldo() {
		return saldo + limite;
	}

//Gets e Sets

 }
```

Agora vamos implementar a classe *ContaPoupanca*:

```
public class ContaPoupanca extends Conta {

	@Override
	public double verificarSaldo() {
		return saldo;
	}
}
```

A classe *ContaPoupanca* é obrigada a implementar o método *verificarSaldo,* pois ela é concreta. Porém, ela não foi obrigada a implementar o método *sacar*, pois esse método não é abstrato e está implementado na superclasse Conta. Dessa forma, a classe *ContaPoupanca* possui o mesmo método *sacar* da classe *Conta* e implementa um comportamento específico no método *verificarSaldo.*

Abaixo está outro exemplo de representação de um diagrama de classes:

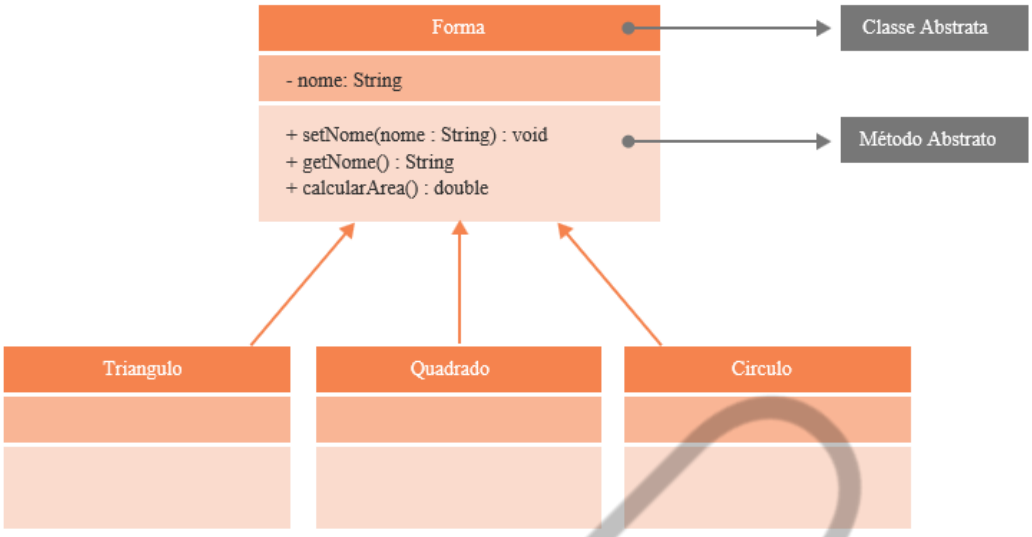

Figura 6.9 – Representação de um diagrama de classes
Fonte: Elaborado pelo autor (2017)

A classe *Forma* é abstrata e possui um atributo com os métodos get e set. Ela define também um método abstrato *calcularArea.* Esse método é abstrato porque a classe *Forma* não consegue calcular a área, pois isso depende do tipo da forma. Assim, todas as subclasses deverão implementar o método abstrato e calcular a sua área.

6.9 Modificador final

O modificador **final** trabalha de forma contrária em relação ao modificador **abstract.** Uma classe Java marcada como final não pode ser estendida. Exemplo:

```
public final class ContaPoupanca extends Conta {

        @Override
        public double verificarSaldo() {
              return saldo;
        }

}
```

A classe *ContaPoupanca* não pode possuir nenhuma subclasse. Dessa forma, não podemos utilizá-la como base para criar nenhuma outra classe:

```
public class ContaPoupancaEspecial extends ContaPoupanca{//Não
Compila
}
```

O código acima não compila, pois a *ContaPoupanca* está marcada como final e por isso não pode ser estendida.

Além das classes, também podemos utilizar a palavra-chave **final** em atributos. Dessa forma, o valor do atributo será imutável, ou seja, não podemos alterar o valor do atributo durante o ciclo de vida do objeto.

O atributo final é chamado de atributos constantes e sua inicialização é feita no momento da declaração:

```
public class Circulo {

        private final double NUMERO_PI = 3.1416;

}
```

Observe que o nome do atributo está um pouco diferente do utilizado até o momento. Por convenção, os atributos constantes devem ser criados com todas as letras em maiúsculas e caso o nome seja composto por mais de uma palavra, estes devem ser separados pelo caractere *underline.*

O atributo NUMERO_PI está marcado como *private*, por isso só é visível dentro da própria classe. Podemos criar um método dentro da classe *Circulo* para calcular o valor da sua respectiva área. Para isso, precisamos de um atributo para armazenar o valor do raio do círculo:

```
public class Circulo {

    private final double NUMERO_PI = 3.1416;

    private double raio;

    public double calcularArea(){
        return NUMERO_PI*raio*raio;
    }

    public double getRaio() {
        return raio;
    }

    public void setRaio(double raio) {
```

```
            this.raio = raio;
        }

    }
```

Observe que utilizamos o atributo final da mesma forma que utilizamos os outros atributos. A diferença é que não podemos alterar o seu valor:

```
    public double calcularArea(){
            raio = 10; //OK
            NUMERO_PI = 10; //Não Compila
            return NUMERO_PI*raio*raio;
}
```

Além de classes e atributos, podemos utilizar o modificador **final** também em métodos, Dessa forma, método não poderá ser sobrescrito:

```
    public final double calcularArea(){
            return NUMERO_PI*raio*raio;
}
```

No código acima, o método *calcularArea* foi marcado como **final**, dessa forma, se criarmos uma subclasse da classe Circulo e tentar implementar um método para sobrescrever o método *calcularArea*, este não irá compilar:

```
public class Circulo2 extends Circulo {

            public final double calcularArea(){//Não compila
            }

}
```

Portanto, o modificador **final** é importante quando queremos que nenhum outro desenvolvedor estenda a nossa classe ou modifique o comportamento de um método em uma classe filha. É utilizado também para criar valores constantes.

O quadro abaixo resume as principais características da palavra-chave final, dependendo de onde está sendo utilizada:

Elemento	Comportamento
Classe	A classe não poderá ser estendida, ou seja, não possuirá subclasses. A classe String, por exemplo, é final.
Método	O método não poderá ser sobrescrito.
Atributo	O valor do atributo definido na sua declaração não poderá ser alterado.

Quadro 6.3 – Principais características da palavra-chave final
Fonte: Elaborado pelo autor (2017)

6.10 Modificador static

Outro modificador presente na plataforma java é o **static.** Esse modificador pode ser aplicado aos membros de uma classe: métodos e atributos.

Um atributo ou método estático são compartilhados por todas as instancias de uma determinada classe. Existem valores de atributos que devem ser compartilhados entre todas os objetos de uma Classe, dessa forma, podemos marcá-la com o modificador **static.**

Quando um atributo é declarado como estático, ele passa a se referir a Classe e não mais à instância da Classe, ou seja, o atributo será igual para todos os objetos, independentemente dos respectivos números de instâncias.

Por isso, se um objeto mudar o valor do atributo estático, todos os outros objetos terão acesso ao novo valor:

```
public class AcessoCatraca {

        private static int totalAcesso;
}
```

Um método marcado como estático não depende das características de cada objeto. Podem ser invocados sem precisar de uma instancia da classe e são utilizados para realizar uma tarefa comum para todos os objetos.

Já utilizamos um método estático. Lembra-se do método *main*?

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

```
}
```

Um método estático utiliza apenas informações contidas em seus parâmetros e nos atributos estáticos, ou seja, um método estático pode acessar somente atributos estáticos, os atributos não estáticos não são acessíveis. O contrário é possível, acessar um atributo estático dentro de um método não estático:

```
public class AcessoCatraca {

    private static int totalAcesso;

    private String nome;

    public void entrar(String nome){
        this.nome = nome;
        totalAcesso = totalAcesso + 1;
    }

    public static int recuperarTotal(){
        return totalAcesso;
    }
}
```

O exemplo acima define um atributo estático (totalAcesso) e um atributo não estático (nome). O método *entrar* depende do atributo nome. Dessa forma, o método não pode ser estático, pois ele precisa utilizar um atributo não estático. Já o método *recuperarTotal* é estático pois utiliza somente o atributo *totalAcesso* que também é estático. Assim, ela não depende de nenhum atributo de instância da classe.

Um detalhe é que se retirarmos o modificador *static* do método *recuperarTotal* não causaria nenhum problema de compilação. Porém a melhor opção seria deixar o método estático, pois ele poderá ser acessado sem uma instância da classe por realizar uma tarefa comum a todos os objetos.

Um exemplo de utilização de métodos estáticos é a classe **java.lang.Math.** Ela possui vários métodos estáticos que auxiliam o desenvolvedor nas operações matemáticas. Veja mais na documentação da Oracle.

Para utilizar um método estático não é preciso de uma instância da classe:

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

```
        int total = AcessoCatraca.recuperarTotal();
        System.out.println("Total " + total);

        long numero = Math.round(2.9);
        System.out.println("Número arredondado: " + numero);
}
```

O código acima acessa o método estático *recuperarTotal*, da classe *AcessoCatraca* e utiliza um método de arredondamento da classe *Math.* Em ambos os casos, não foi preciso instanciar suas respectivas classes.

Para acessar o método *entrar***,** da classe *AcessoCatraca* será preciso criar o objeto:

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

        AcessoCatraca a1 = new AcessoCatraca();
        a1.entrar("Thiago");

        AcessoCatraca a2 = new AcessoCatraca();
        a2.entrar("Leandro");

        int total = AcessoCatraca.recuperarTotal();
        System.out.println("Total " + total);

}
```

Qual será o resultado da execução acima?

O método *entrar* incrementa o valor do atributo estático *totalAcesso,* um atributo estático se refere à classe e não ao objeto, então eles irão compartilhar o mesmo valor. Como o método foi chamado duas vezes, independentemente do objeto, no final da execução o valor total será 2:

```
Total 2
```

É possível também chamar o método estático utilizando o objeto, porém o resultado final será igual:

```
int total = a1.recuperarTotal();
System.out.println("Total " + total);
```

Resultado:

```
Total 2
```

Os métodos e atributos estáticos em um diagrama de classe são representados com um sublinhado:

Figura 6.10 – Os métodos e atributos estáticos em um diagrama de classe representados com um sublinhado
Fonte: Elaborado pelo autor (2017)

6.11 Constantes

É comum utilizar os modificadores **public, static** e **final** ao declarar uma constante no Java. As constantes podem ser públicas, ou seja, todos podem ter acesso. Estático, pois não existe a necessidade de cada objeto ter a sua própria constante e final, para que o valor da constante nunca se altere.

Por convenção, o nome de uma constante é sempre escrito em maiúsculo com as palavras separadas por *underline:*

```
public class Constantes {

     public static final String JANEIRO = "Janeiro";
     public static final double TAXA_RETIRADA = 10;
     public static final int NUMERO_DIAS_SEMANA = 7;
     public static final Estado SAO_PAULO = new Estado("São Paulo","SP");

}
```

Para acessar as constantes, basta utilizar o nome da classe*:*

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

          System.out.println(Constantes.JANEIRO);
```

```
            System.out.println(Constantes.TAXA_RETIRADA);
        }
```

6.12 Interfaces:

A plataforma Java e algumas outras linguagens de programação orientada a objetos possuem o conceito de **Interface**. Interface define um conjunto de requisitos para as classes implementá-las. Uma interface não é uma classe.

Interface é um contrato entre a classe e o mundo externo. Quando uma classe implementa uma interface, ela está comprometida a fornecer todos os comportamentos definidos na interface.

Uma interface em Java não pode ser instanciada. Assim como as classes, uma interface pode ser composta por atributos e métodos. Porém, como ela não é instanciada, não apresenta construtores.

A plataforma Java não permite a herança múltipla, ou seja, não é possível herdar duas classes diretamente. Porém, uma classe Java pode implementar uma ou mais interfaces, devendo, assim, definir todos os métodos publicados por todas as interfaces implementadas.

Uma classe abstrata não é obrigada a implementar todos os métodos definidos na interface. Porém, em uma classe concreta é obrigatório.

Todos os atributos e métodos de uma interface são implicitamente públicos. Todo atributo em uma interface é implicitamente público, final e estático (Constante). Como esses qualificadores são fixos, não precisamos declará-los.

Exemplo de uma interface:

```
public interface Autenticavel {

            String MSG_LOGOUT = "Saindo";

            boolean login(String usuario, String senha);

            void logou();
```

```
}
```

Note que após o modificador de acesso, a palavra-chave utilizada é ***interface*** e não ***class.*** Dessa forma, sabemos que se trata de uma interface e não de uma classe. Observe também que o arquivo gerado possui a extensão .java.

Para criar uma interface no eclipse, clique com o botão direito do mouse na pasta **src** e escolha a opção **New** -> **Interface**:

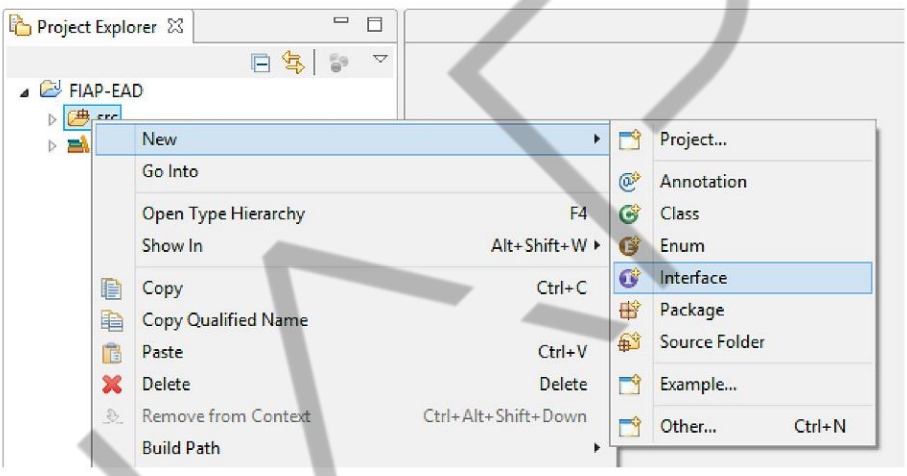

Figura 6.11 – Exemplo de criação de uma interface no eclipse
Fonte: Elaborado pelo autor (2017)

Após isso, defina um pacote e nome da interface:

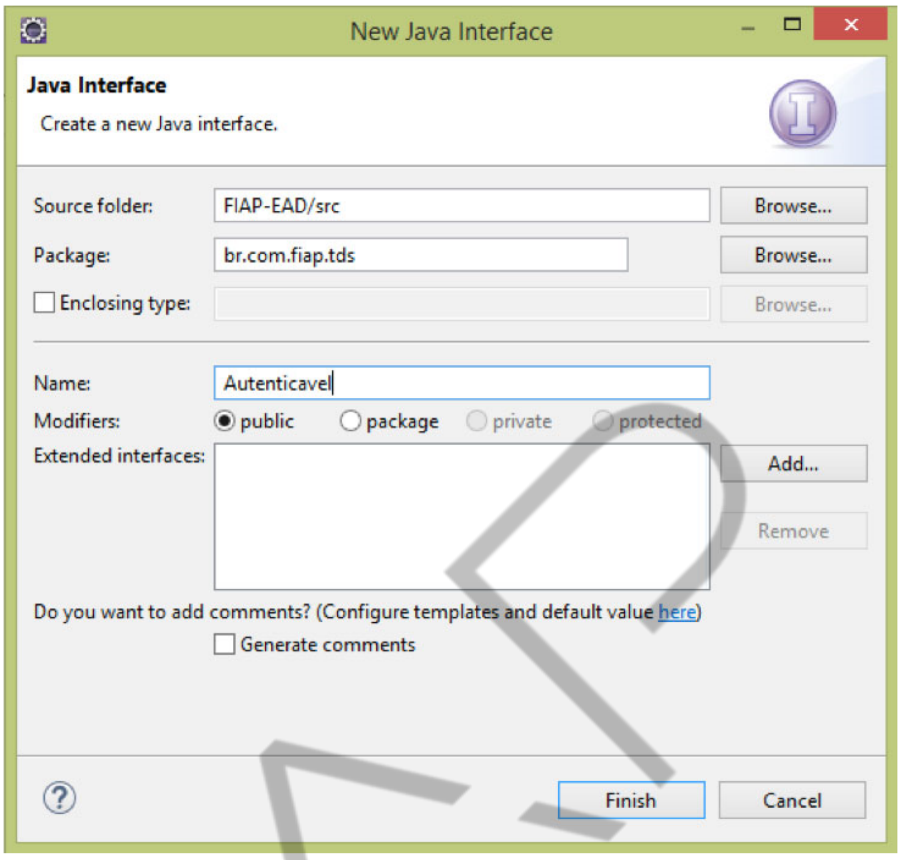

Figura 6.12 – Definindo pacote e nome
Fonte: Elaborado pelo autor (2017)

A figura abaixo representa uma interface no diagrama de classes:

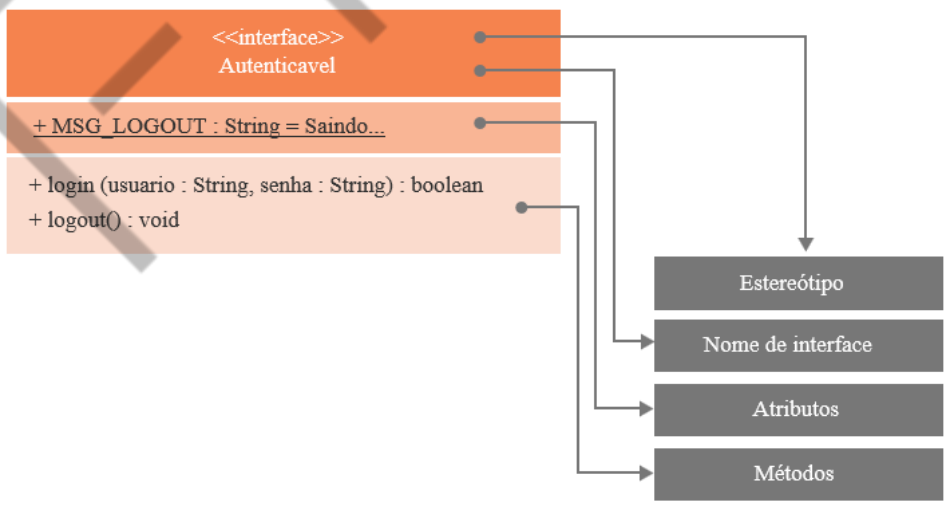

Figura 6.13 – Representação de uma interface no diagrama de classes
Fonte: Elaborado pelo autor (2017)

Para uma classe implementar uma interface, devemos utilizar a palavra-chave **implements:**

```
public class Usuario implements Autenticavel{

	@Override
	public boolean login(String usuario, String senha) {
		// TODO Auto-generated method stub
		return false;
	}

	@Override
	public void logou() {
		// TODO Auto-generated method stub

	}

}
```

Como a classe *Usuario* implementa a interface *Autenticavel,* ela deve implementar todos os métodos definidos pela interface. Caso isso não seja feito, teremos um erro de compilação.

Se uma classe implementar mais de uma interface, devemos separá-las por vírgula:

```
public class Usuario implements Autenticavel, Serializable {
		// Código…
}
```

E se uma classe implementar uma ou mais interface e estender uma classe, primeiro precisamos definir a herança:

```
public class Usuario extends Pessoa implements Autenticavel {
		// Código…
}
```

No diagrama de classes, a representação de uma implementação é definida por uma seta com a sua linha pontilhada:

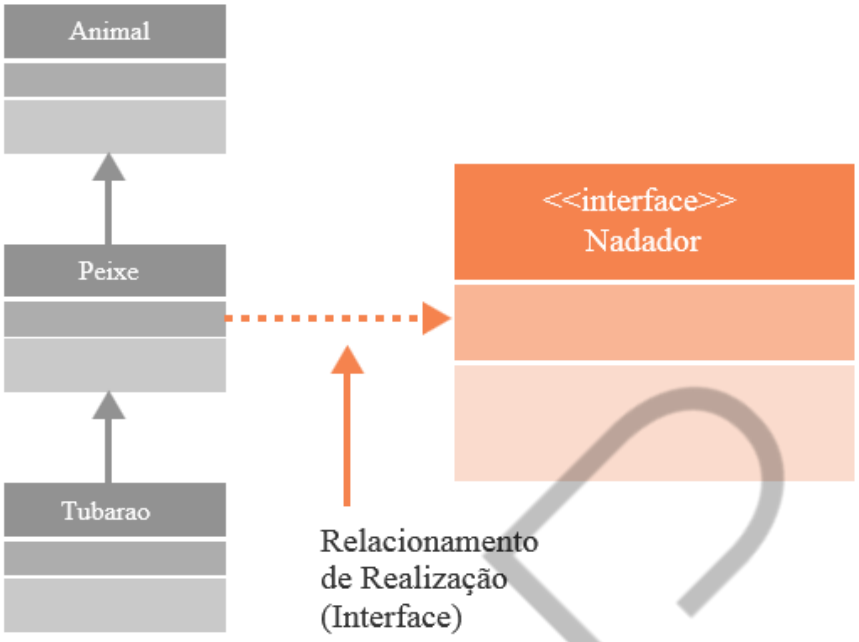

Figura 6.14 – A representação de uma implementação
Fonte: Elaborado pelo autor (2017)

Sempre utilize interfaces quando for preciso prover operações comuns a classes de hierarquia diferentes:

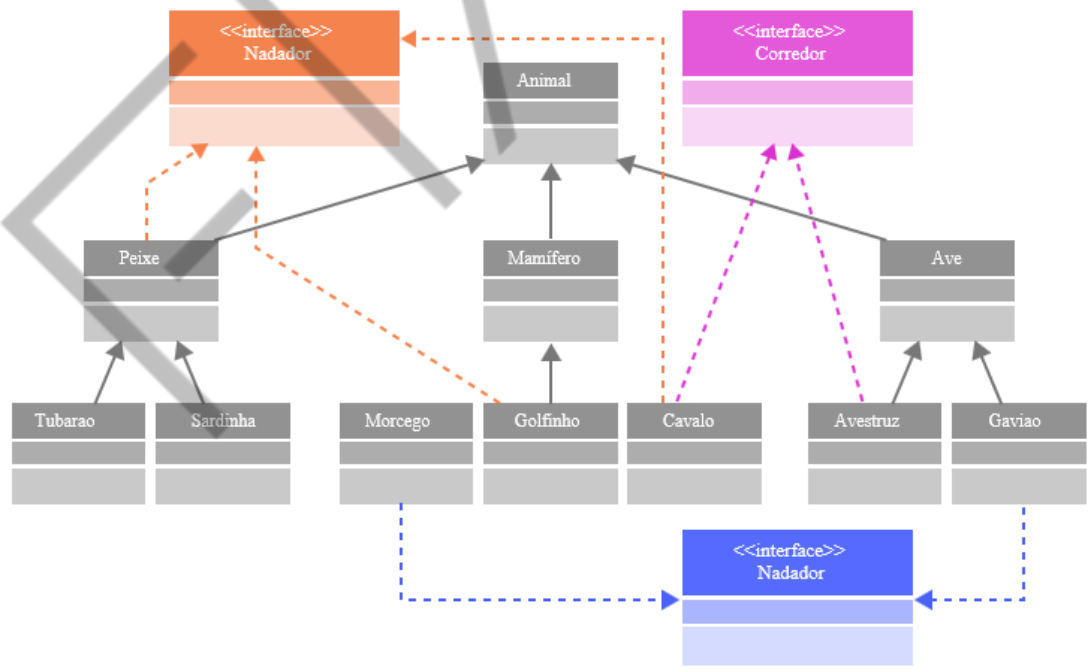

Figura 6.15 – Utilização de interfaces
Fonte: Elaborado pelo autor (2017)

O diagrama acima, define que as classes *Morcego* e *Gaviao* implementarão a interface *Voador,* pois ambas as classes possuem essas características. Já a classe *Cavalo e Avestruz* são *Corredores.* E por fim, todas as classes que definem um animal Nadador deverão implementar essa interface.

Utilizar interfaces permite o uso do polimorfismo. Por exemplo, duas classes que implementam uma mesma interface podem ser atribuídas a uma variável do tipo da Interface. Porém, a execução dependerá do tipo do objeto que essa variável armazena:

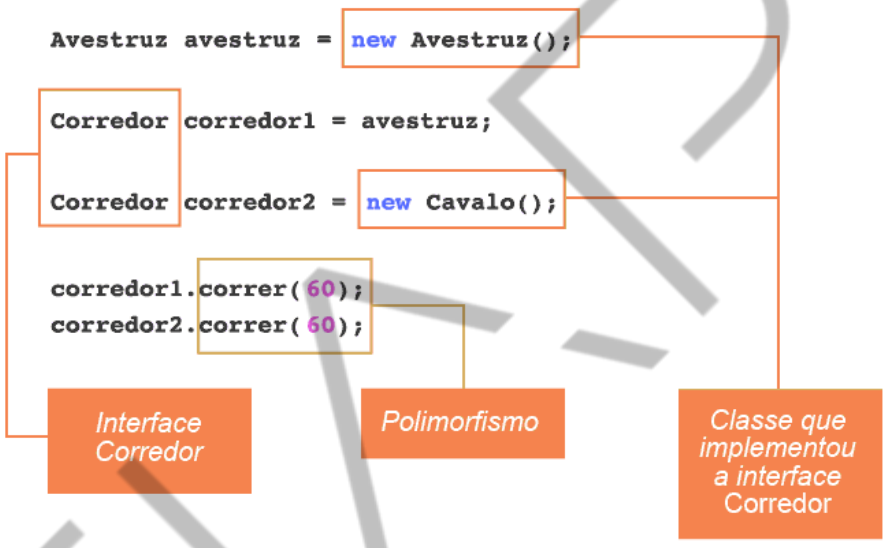

Figura 6.16 – Exemplo de duas classes que implementam uma mesma interface
Fonte: Elaborado pelo autor (2017)

No exemplo acima, as classes *Avestruz* e *Cavalo* implementam a interface *Corredor*. Foi instanciada cada uma dessas classes e atribuídas a duas variáveis do tipo da interface (Corredor). Quando o método *correr* é invocado, será executado o método definido na classe correspondente ao tipo do objeto armazenado na variável.

Uma interface pode estender uma ou mais interfaces:

```
public interface Gerenciavel extends Autenticavel {
}

public interface Gerenciavel extends Autenticavel, Serializable
{
}
```

Podemos ter métodos concretos em uma interface. Para isso, devemos utilizar o modificador *default.* Esses método chamado de **default method**, serão herdados por todas as classes que implementarem a interface. Não sendo necessário implementá-la na classe.

Exemplo:

```
public interface Corredor {

		default void parar(){
			System.out.println("Parando..");
		}

}
```

Dessa forma, as classes que implementarem a interface *Corredor* já receberão o método *parar*, sem precisar implementá-lo.

```
public class Avestruz implements Corredor{

}
```

Podemos utilizar o método com uma instancia da classe:

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

		Corredor corredor = new Avestruz();
		corredor.parar();

}
```

Um método **static** é outro método que possui implementação em uma interface. A diferença entre o método default e o método estático é que o estático pertence à interface e não pode ser sobrescrito. Um **default method** pode ser sobrescrito em uma classe e precisa de uma de suas instâncias para ser executada.

```
	public interface Corredor {

		default void parar(){
			System.out.println("Parando..");
		}

		static void acelerar(){
```

```
            System.out.println("Acelerando");
        }
}
```

Na classe avestruz, podemos sobrescrever o método default:

```
public class Avestruz implements Corredor{

        @Override
        public void parar() {
            System.out.println("Avestruz parando...");
        }
}
```

Porém, não é possível sobrescrever o método default. E para executar o método estático, devemos referenciar a interface e não a classe ou o objeto dela:

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

        Corredor corredor = new Avestruz();
        corredor.parar();

        Corredor.acelerar();

}
```

Em uma interface, podemos definir todos os tipos de métodos em conjunto, ou seja, podemos ter métodos abstratos (sem implementação, que deverá ser implementado na classe), métodos defaults e métodos estáticos:

```
public interface Corredor {

        void correr(int velocidade);

        default void parar(){
            System.out.println("Parando..");
        }

        static void acelerar(){
            System.out.println("Acelerando");
        }
}
```

Para finalizar, podemos utilizar o operador **instanceOf** para verificar se uma classe é do tipo de uma interface ou outra classe (herança). Esse operador retorna **true** se o objeto à esquerda do operador é do tipo da classe ou interface especificada à direita do operador:

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

		Corredor avestruz = new Avestruz();

		if (avestruz instanceof Corredor){
			System.out.println("É um corredor");
		}else{
			System.out.println("Não é um corredor");
		}

}
```

No exemplo acima, o resultado será “É um corredor”, pois o objeto referenciado na variável avestruz é do tipo Avestruz, e este implementa a interface *Corredor.*

Podemos utilizar esse operador também para verificar se o objeto é do tipo de uma classe:

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

		Corredor avestruz = new Avestruz();

		if (avestruz instanceof Animal){
			System.out.println("É um animal");
		}else{
			System.out.println("Não é um animal");
		}

}
```

Neste caso, a classe *Avestruz* herda da classe *Animal.* Dessa forma, o resultado final será “É um animal”. Com esse operador, podemos testar um tipo de um objeto.

REFERÊNCIAS

BARNES, David J. **Programação Orientada a Objetos com Java:** Uma introdução Prática Utilizando Blue J. São Paulo: Pearson, 2004.

CADENHEAD, Rogers; LEMAY, Laura. **Aprenda em 21 dias Java 2 Professional Reference**. 5.ed. Rio de Janeiro: Elsevier, 2003.

DEITEL, Paul; DEITEL, Harvey. **Java Como Programar.** 8.ed. São Paulo: Pearson, 2010.

HORSTMANN, Cay; CORNELL, Gary. **Core Java:** Volume I Fundamentos. 8.ed. São Paulo: Pearson 2009.

SIERRA, Kathy; BATES, Bert. **Use a cabeça! Java.** Rio de Janeiro: Alta Books, 2010.