PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Informática e Ciências Exatas - Curso de Sistemas de Informação
Laboratório de Programação Orientada por Objeto - Professor: Paulo Amaral

RELATÓRIO 2 (2/2015) - TRABALHO EM GRUPO

Métodos construtores, herança, fluxos e arquivos

1. ESPECIFICAÇÕES

Deve ser **entregue** um texto com o **relatório completo** do trabalho que conste na capa (1ª. página) os **nomes completos de todos os componentes do grupo**. O número de componentes do grupo será especificado pelo professor durante a aula, não sendo aceitos trabalhos anônimos.

No **relatório** deve constar tudo que foi especificado no exercício e os c**ódigos-fonte de todos os programas** envolvidos nos exercícios**.**

Todos programas devem **imprimir o nome completo de todos os componentes do grupo na tela** e os **arquivos fontes dos programas, além de serem incluídos no texto do relatório, devem ser entregues em arquivos separados no SGA**, devendo conter o **cabeçalho completo** com entrada, saída, descrição e nome completo de todos os componentes do grupo (não sendo aceitos trabalhos sem os nomes. **Cópias grosseiras** serão desconsideradas, ou seja, a **nota será igual a 0 (zero).**

A lista de exercícios tem partes que devem ser lidas (**Leitura**), outras (**Prática)** que devem ser exercitadas em casa e/ou nos laboratórios de informática e finalmente alguns **exercícios** (**Entregar) que devem ser entregues com o relatório.**

Haverá apresentação oral dos trabalhos

2. CLASSES E MÉTODOS EM JAVA e C#

(Baseado no material dos Profs João Caram, Hugo Bastos, Anna Tostes e

da Apostila de C# e Orientação a Objetos da K19 Treinamentos)

2.1 Introdução (**Leitura)**

Conforme já foi visto nos exercícios do relatório anterior, uma **classe em Java** tem dois componentes principais: a **declaração** da classe e o **corpo** da classe (definição). A declaração da classe é a primeira linha de código em uma classe. Ela deve, no mínimo, declarar o nome da classe.

O corpo da classe segue a declaração da classe e aparece entre chaves. O corpo da classe possui a declaração de seus **atributos** (fornecem seu estado) e dos **métodos** (que implementam seu comportamento). Os atributos e métodos da classe são chamados de **membros** da classe.

Membros públicos são os campos e funções membro que são visíveis externamente à classe. O conjunto de membros públicos de uma classe formam a **interface pública** da classe. Em Java, todos os membros são publicados para todas as classes de um mesmo pacote, a não ser que se defina o contrário.

Os métodos, assim como os atributos, devem ser introduzidos dentro do corpo da classe.

Java utiliza semântica por referência para tratar objetos de classe. Isto significa que uma declaração de objeto na verdade é uma declaração de uma referência para um objeto. Uma referência é um ponteiro (apontador) constante.

Referências são chamadas de alias (sinônimos), uma vez que elas simplesmente dão nome a um objeto existente na memória.

A declaração de uma referência não implica na criação do objeto. O objeto será construído apenas se utilizarmos a cláusula new.

Ao contrário dos ponteiros as referências possuem apenas dois estados: ou ela está apontando para um objeto do tipo declarado pela referência ou ela possui o valor null.

2.2 Exemplo1: A Classe Date1 em Java (prática):

```
class Date1 { // arquivo fonte disponível no SGA
        public int dia, mes, ano ; // tres atributos inteiros
        public void iniciaData (int d ,int m ,int a ) {
                if ( a > 1900)
                        ano = a ;
                if ( (m > 0) && (m <= 12) )
                        mes = m;
                if ( ( d > 0) && (d <= 31) )
                        dia = d ;
        }
        public boolean anoBisexto ( ) {
                if ( ano % 4 != 0)
                        return false ;
                else if ( ano % 100 != 0)
                        return true ;
                else if ( ano % 400 != 0)
                        return false ;
                else return true ;
        }


        // Programa de teste  Date1
        public static void main( String rgs [ ] ) {
                Date1 d = new Date1( );
                d.iniciaData (4 , 10 , 2000);
                System.out .println (d. dia + "/" + d.mes + "/" + d. ano );
                if (d. anoBisexto ( ))
                        System.out.println ("Ano eh Bisexto" );
                else
                        System.out.println ("Ano nao eh Bisexto" );
        }
}
```

Em Java, uma instância de uma classe é interpretada como uma referência para um objeto e não o objeto propriamente dito. Para se criar um objeto deve-se declarar uma referência para o objeto e, em seguida, se construir o objeto com a cláusula new.

Declaração de um Objeto:

- nomeClasse nomeObjeto;
- Cria-se a referência mas não se cria o objeto.
- Criação do objeto através da cláusula new.

- Uma vez criado o objeto, a referência não pode ser manipulada numericamente: **Date1 d**;

- Cria-se uma referência para um objeto do tipo Date1, mas não se aloca a memória para armazenar o objeto e a variável d aponta para NADA (null)

- Para criar efetivamente o objeto Date1 e fazer com que a referência aponte para Date1:
  d = new Date1( );

Objetos são instâncias de uma classe. Define-se instância como sendo um elemento com o tipo da classe e um estado corrente individual. Por exemplo:

Classe: Date (tipo com dia, mês e ano)
Objeto de Date: d = 04/10/2000;

2.3 Métodos construtores (leitura):

Uma classe pode possuir também métodos especiais denominados construtores.

Um **construtor** é um método especial que possui o mesmo nome da classe e não possui tipo de retorno, tendo por finalidade iniciar os atributos do objeto.

Um **Construtor default** é um construtor sem parâmetros cuja finalidade é inicializar objetos com valores padrão.

Ao se construir um objeto de uma classe, sua memória é inicializada. Se não for definido um modo de inicialização para o objeto, a linguagem Java utilizará valores padrão (zero para tipos numéricos e null para referências). Por exemplo, se um objeto da classe Date for construído com construtores padrões tem-se:

d = new Date ( ); // cria d = 0/0/0000

Construtores são usados para inicializar objetos com valores diferentes do padrão. Para se declarar construtores, eles devem ter as seguintes características: devem possuir o mesmo nome da classe; não devem possuir valores de retorno; e uma classe pode ter de 0 a muitos construtores.

Diversas são as razões para se utilizar construtores:

- Algumas classes não possuem estado inicial aceitável sem parâmetros. Desta forma, o construtor “obriga” o programador a inicializar o objeto assim que ele é criado, atribuindo um valor que seja semanticamente válido, em função da especificação do sistema. Esta prática garantirá ao programa maior robustez contra esquecimento humano.

- Por outro lado, muitas vezes, fornecer um estado inicial pode ser conveniente e aceitável quando da construção de alguns tipos de objetos. O construtor facilita a atribuição de valores em bloco aos atributos de um objeto, aumentando a legibilidade do código e a produtividade do programador.

- Construir um objeto aos poucos pode ser desgastante de forma que pode ser conveniente que se tenha um estado inicial correto quando forem criados. O construtor irá aumentar a robustez do código contra possíveis esquecimentos do programador, especialmente quando o objeto em questão possui um número grande de atributos.

2.4 Exemplo2: A Classe Date2 em Java (prática):

```
class Date2 { // arquivo fonte disponível no SGA
        public int dia, mes, ano ; // tres atributos inteiros
        Date2(  ) { // construtor default sem parâmetros
                }
        Date2(int d, int m, int a) {// construtor com parâmetros passados por valor
                dia = d; mes = m; ano = a;
        }

        public boolean anoBisexto ( ) {
                if ( ano % 4 != 0)
                        return false ;
                else if ( ano % 100 != 0)
                        return true ;
                else if ( ano % 400 != 0)
                        return false ;
                else return true ;
        }



        // Programa de teste Date2
        public static void main( String args [ ] ) {
                Date2 d1 = new Date2 ( ) ;
                Date2 d2 = new Date2 (20, 2, 2015) ;
                System.out.println ("Data com valores padrao: " + d1.dia + "/" + d1.mes + "/" +
                                        d1.ano ) ;
                if (d1.anoBisexto ( ))
                        System.out.println ("Ano eh Bisexto" );
                else
                        System.out.println ("Ano nao eh Bisexto" );

                System.out.println ( "Data LAB2: " + d2.dia + "/" + d2.mes + "/" + d2.ano ) ;
```

```
            if (d2.anoBisexto ( ))
                System.out.println ("Ano eh Bisexto" );
            else
                System.out.println ("Ano nao eh Bisexto" );

        }


}
```

2.5 Encapsulamento: ocultar informações (leitura)

Encapsulamento consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente apenas os métodos que alteram estes estados. Exemplo: para usar o aparelho de telefone, não é necessário conhecimento dos circuitos internos do aparelho ou como a rede telefônica funciona, basta saber manipular os botões e o fone.

O encapsulamento permite que os dados sejam protegidos de acesso ilegal. Uma classe pode ter qualquer quantidade de dados e métodos. Em geral, deseja-se ocultar determinados dados e/ou métodos do cliente que usará uma classe. Do ponto de vista teórico, uma classe é um Tipo Abstrato de Dados, e como tal, deve produzir um novo tipo mais genérico (abstrato), que deve prover um serviço de alto nível, ao passo que deve ocultar detalhes de implementação.

Por Exemplo: ao acessar o campo dia da classe Date e alterar um valor, digamos, de 20 para 30, uma data 20/02/2010 pode se tornar inválida, uma vez que não existe a data 30/02/2010. Para evitar este tipo de problema deve-se encapsular (esconder) os dados, e prover uma interface pública pequena que permita alterar estes dados de forma consistente. Ou seja, apenas a própria classe Date precisa ter conhecimento de suas especificações, e deve prover mecanismos corretos e robustos para a sua manipulação.

O encapsulamento de atributos e métodos é feito através de palavras reservadas da linguagem que chamadas de modificadores de acesso. A linguagem Java possui 4 modificadores de acesso:

public: membros são acessíveis de qualquer lugar do programa.
private: membros são acessíveis apenas na própria classe.
protected: membros são acessíveis à classe e por suas subclasses. O modificador protected deixará visível o atributo para todas as suas subclasses e classes que pertencem ao mesmo pacote. A principal diferença é que apenas classes do mesmo pacote tem acesso ao membro. O pacote da subclasse não tem acesso ao membro.
Pacote (friendly): Membros declarados sem modificador de acesso são acessíveis apenas às classes dentro do mesmo pacote onde ela se encontra.

A linguagem C# também possui 4 modificadores de acesso:

private: somente a classe incluída pode acessar o objeto, método ou variável.
public: todos têm acesso**.**
protected: visíveis apenas para as classes derivadas por meio de herança.

internal: acesso livre somente dentro de um assembly (DLL ou EXE) correspondente ao JAR do Java. Fora do assembly da classe é inacessível.

2.6 Exemplo3: A Classe Date3 em Java (prática):

```
class Date3 { // arquivo fonte disponível no SGA
        private int dia, mes, ano ; // tres atributos inteiros
        Date3( ) { // construtor default sem parâmetros inicializados com valores validos
                dia = 1 ; mes = 1 ; ano = 1900;
        }
        public Date3(int d, int m, int a) {// construtor com parâmetros passados por valor
                if ( ( d > 0) && (d <= 31) )
                        dia = d ;
                if ( (m > 0) && (m <= 12) )
                        mes = m;
                if ( a > 1900)
                        ano = a ;
        }

        public boolean anoBisexto ( ) {
                if ( ano % 4 != 0)
                        return false ;
                else if ( ano % 100 != 0)
                        return true ;
                else if ( ano % 400 != 0)
                        return false ;
                else return true ;
        }

        public int getDia ( ) { return dia ; }
        public int getMes ( ) { return mes ; }
        public int getAno ( ) { return ano ; }

        // Programa de teste Date3
        public static void main( String args [ ] ) {
                Date3 d1 = new Date3 ( ) ;
                Date3 d2 = new Date3(19, 8, 2014) ;
                System.out.println ( "Data padrao: " + d1.getDia( ) + "/" + d1.getMes( ) + "/" +
                                        d1.getAno( )) ;
                System.out.println ( "Data LAB2: " + d2.getDia( ) + "/" + d2.getMes( ) + "/" +
                                d2.getAno( ) ) ;
        }

}
```

2.7 Membros estáticos

Membros estáticos são atributos ou métodos comuns a todos os objetos de uma classe. Normalmente, um atributo declarado em uma classe é um membro de instância dessa classe. Ou

seja, o sistema cria uma cópia deste atributo para cada instância da classe declarada pelo programa. Cada cópia deste atributo terá seu próprio valor, e o conjunto de valores dos atributos de um objeto definirá o estado deste objeto.

Um atributo de classe declarado com static, por outro lado, é único para todas as instâncias da classe. O sistema aloca memória apenas para um valor, independentemente do número de instâncias da classe criadas. Todas essas instâncias compartilham esse atributo. Seu funcionamento é análogo ao de uma variável global dentro de uma classe. Ou seja, o primeiro objeto inicializa a variável; e os objetos subsequentes compartilham a mesma variável.

Muitas vezes, atributos estáticos são usados para comunicação entre objetos de uma mesma classe. Membros estáticos são extremamente úteis para se implementar contadores de acesso ou identificadores de autoincremento.

Por exemplo, suponha que está sendo desenvolvida uma aplicação de bate papo. Cada usuário que se conecta a uma sala de bate papo, faz com que o servidor crie um objeto da classe ConexaoCliente para tratar as mensagens do usuário conectado. Suponha que seja necessário limitar o número de usuários simultâneos em 100 (por questões de desempenho e memória). A classe ConexaoCliente pode ter um atributo inteiro chamado contador que é estático e é incrementado no construtor da classe e decrementado no destrutor da classe. Desta maneira, cada vez que um objeto é construído o número de conexões simultâneas é aumentado.

Membros estáticos também podem ser usados para definir constantes. Como a variável é compartilhada por todos os objetos de uma classe, a utilização de membros estáticos constantes pode permitir grande economia de memória.

2.8 Exemplo 4: A Classe Date4 em Java (prática):

```
class Date4{ // arquivo fonte disponível no SGA
        private int dia, mes, ano ; // tres atributos inteiros
        private static int numDias [ ] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        public Date4( ) { // construtor default sem parâmetros inicializados com valores validos
                dia = 1 ; mes = 1 ; ano = 1900;
        }

        public Date4 ( int dia , int mes, int ano ) {// construtor com parâmetros passados por valor
                if ((mes > 0) && (mes <= 12)) this.mes = mes;
                else this.mes = 1 ;
                if (( dia > 0) && (dia <= numDias[mes - 1])) this.dia = dia ;
                else this.dia = 1 ;
                if ( ano > 1900) this.ano = ano ;
                else this.ano = 1900;
        }



        public boolean anoBisexto ( ) {
                if ( ano % 4 != 0)
                        return false ;
```

```
            else if ( ano % 100 != 0)
                    return true ;
            else if ( ano % 400 != 0)
                    return false ;
            else return true ;
    }

    public int getDia ( ) { return dia ; }
    public int getMes ( ) { return mes ; }
    public int getAno ( ) { return ano ; }


    // Programa de teste Date4
    public static void main( String args [ ] ) {
            Date4 d1 = new Date4 ( ) ;
            Date4 d2 = new Date4(20, 2, 2015);
            Date4 d3 = new Date4(29, 2, 2014) ;
            Date4 d4 = new Date4(30, 01, 1899) ;
            System.out.println ( "\nData padrao: " + d1.getDia( ) + "/" + d1.getMes( ) + "/" +
                                    d1.getAno( )) ;
            System.out.println ( "Data LAB2: " + d2.getDia( ) + "/" + d2.getMes( ) + "/" +
                                    d2.getAno( ) ) ;
            System.out.println ( "Data errada 1 alterada: " + d3.getDia( ) + "/" + d3.getMes( ) +
                                    "/" + d3.getAno( )) ;
            System.out.println ( "Data errada 2 alterada: " + d4.getDia( ) + "/" + d4.getMes( ) +
            "/" + d4.getAno( ) ) ;
    }

}
```

3. EXERCÍCIOS PARA ENTREGAR

3.1 A partir dos exemplos 1 a 4 (apresentados nas seções anteriores), fazer **programas** (codificado inicialmente em **Java** e **depois em C#)** que implemente a **classe Date5** com as seguintes especificações **(Exercício para entregar)**:

3.1.1 A **classe Date5** suporta as seguintes operações:

a) construtor: recebe os valores de dia, mês e ano como parâmetros e inicia a data;
b) getters: métodos para retornar os valores de dia, mês e ano;
c) setters: métodos para alterar os valores de dia, mês e ano;
d) extenso: retorna a data por extenso como uma string;
e) proximoDia: altera a data internamente de modo que esta passará a representar o dia seguinte de uma data (atenção às mudanças de mês e ano).
f) diffDias: para calcular a diferença em dias entre duas datas
g) dataValida, para verificar a validade da data.
h) Dizer o dia da data no ano (1 a 366)

3.1.2 Para testar, inclua no programa testes:
a) Ler duas datas e calcule o número de dias de uma data para outra. Exemplo de uma saída a ser obtida:
O número de dias entre 2 de Janeiro de 2013 até 9 de Janeiro de 2013 é 8 dias.

b) Crie outros objetos da classe **Date5** e simule seu uso.

3.2 Fazer um **programa** (codificado inicialmente em **Java** e depois em **C#**) que implemente a **classe Conta,** com as seguintes especificações **(Exercício para entregar)**:

3.2.1 Atributos obrigatórios:

```
private  String titular;    // nome do titular da conta
private int agencia;        // número da agência da conta
private int numConta;   // número da conta
private static int cont = 0;   //contador de contas existentes
private int tipoConta;   // indica o tipo da conta:  corrente, poupança, investimento
private double saldo;   // saldo atual da conta
```

3.2.2 Métodos obrigatórios:

```
public Conta(String nome, int ag, int tipo, double saldoInicial); // Contrutor
public Excluir_Conta(int numero_conta); // Contrutor
public double obterSaldo(int numConta);
public void depositar(double credito, int numConta);
public void sacar(double quantia, int numConta);
public void ImprimeSaldo (int numConta);
public void ImprimeContas ();

}
```

3.2.3 Os dados das contas devem ser armazenados em vetor estático do tipo abstrato Conta com limite de 100 clientes, ou seja:

```
const int MAXCONTAS = 100; // número máximo de contas suportado
static Conta[ ] vetContas = new Conta[MAXCONTAS]; //vetor de contas
```

3.2.4 O programa deve apresentar inicialmente na tela um menu com as seguintes opções:

1. Criar um conta nova.
2. Excluir uma conta existente
3. Depositar em uma conta
4. Sacar de em uma conta
5. Imprimir saldo de uma conta
6. Imprimir uma relação das contas existentes informando o número da conta e o nome do titular da conta
7. Sair do programa

3.2.5 O programa deve obter a opção do usuário, chamar o método correspondente, apresentar o resultado e sempre voltar ao menu inicial, exceto quando for selecionada a opção 7 (Sair do programa).

3.2.6 Usar as técnicas de encapsulamento e ocultação explicadas nas aulas teóricas.

3.2.7 Crie outros objetos da classe **Conta** e simule seu uso.

3.3 Fazer um **programa** (codificado inicialmente em **Java** e depois em **C#**) que implemente a **classe Quadrado,** com as seguintes especificações **(Exercício para entregar)**:
3.3.1 Deve conter uma propriedade de instância Lado que possua assessores get e set para dados private.
3.3.2 Forneça dois construtores: um que não receba argumentos e outro que receba um comprimento lado com valor.
3.3.3 Usar as técnicas de encapsulamento e ocultação explicadas nas aulas teóricas.
3.3.4 Escreva uma classe de aplicativo que teste a funcionalidade da classe Quadrado.

4. HERANÇA DE CLASSE

4.1 Introdução (Leitura)

Herança Simples é uma técnica para construir novas classes, chamadas de classes derivadas, a partir das classes já existentes, que são ditas classes base. A herança permite o reuso do comportamento de uma classe na definição de outra. A classe derivada herda todas as características da classe base, podendo adicionar novas características.

Pode-se usar a herança na OOP para classificar os objetos em programas, de acordo com as características comuns e a função. Isso torna mais fácil e claro trabalhar com os objetos. Também torna a programação mais fácil, porque permite combinar as características gerais em um objeto-pai e herde essas características nos objetos-filhos. Por exemplo, pode-se definir um objeto funcionário, que define todas as características gerais do funcionário, mas também adicionar características exclusivas aos gerentes de uma empresa. O objeto gerente refletirá automaticamente qualquer alteração na implementação do objeto funcionário.

A herança permite construir uma hierarquia de classes e classificar tipos da mesma forma que tipos classificam valores. A herança estabelece as relações de especialização e generalização entre classes base e derivadas. Membros da classe base podem ser redefinidos ou estendidos na classe derivada. A herança pode ser analisada como um mecanismo para definição de subtipagem (aspecto de tipos), que permite estender os serviços de um módulo.

A **classe base** também é denominada de classe **genérica**, **super classe** ou **classe mãe**. As **classes derivadas** também são denominada de **específicas**, **sub classes** ou **classes filhas**.

Quando o operador new é aplicado em uma sub classe, o objeto construído possuirá os atributos e métodos definidos na sub classe e na super classe.

Em Java, diz-se que a classe derivada estende a classe base. Com isso, usa-se a **palavra chave extends**. **Em Java, qualquer classe herda da classe Object.**

O **exemplo 5** a seguir exibe uma classe **NovaData,** que estende a estrutura da classe **Date4** (exemplo 4), adicionando a capacidade de gerar uma versão por extensão do estado do objeto.

4.2 Exemplo 5: A Classe NovaData em Java (prática):

class NovaData extends Date4 { **// arquivo fonte disponível no SGA**

```
        private static final String NomeMeses [  ] = {"Janeiro", "Fevereiro", "Março",
        "Abril", "Maio", "Junho", "Julho ", "Agosto" ,
        "Setembro", "Outubro" , "Novembro" , "Dezembro" } ;
        public String toString ( ) {
                String temp = getDia ( ) + " de " + NomeMeses [getMes() - 1]  + " de " + getAno ( ) ;
                return temp ;
        }


public static void main ( String args [ ] ) {
        NovaData d1 = new NovaData ( ) ;
        System.out.println ( "\nData padrao: " + d1.getDia ( ) + "/" +
                d1.getMes ( ) + "/" + d1.getAno ( ) ) ;
        System.out.println( "Data padrao: " + d1.toString ( ) );
        if (d1.anoBisexto ( ))
                System.out.println( "Ano eh Bisexto " ) ;
        else
                System.out.println( "Ano nao eh Bisexto " ) ;
        }
}
```

Note que todos os atributos e métodos existentes na classe **Date4** também estão presentes na classe **NovaData**. Entretanto, os atributos declarados como privados na classe Data não poderão ser acessados diretamente na classe NovaData. Esta proteção irá garantir a robustez do código da classe Date e protegê-lo contra acesso não especializado.

Em **C# todas as classes derivam da classe Object**.

class NomeDaClasse : ClasseBase { ...}

Ou seja, as classes derivadas são vinculadas a classe base utilizando o operador (:). Não é necessário redefinir o conteúdo já declarado na classe base.

As cláusulas this e base são referências que indicam a própria classe e a classe base, respectivamente. Entende-se como classe base, a classe cuja a classe atual herda as propriedades e atributos. Sua notação pode ser observada no seguinte exemplo:

```
this.nomeAtributo = valor ;
valor = this.nomeAtributo ;
this.NomeMetodo ( );
base.nomeAtributoClasseBase = valor ;
valor = base.nomeAtributoClasseBase ;
```

base.NomeMetodoClasseBase ( );

4.3 Exemplo 6: A Classe NovaData em C# (prática):

```
namespace Date { // arquivo fonte disponível no SGA

class Date4{
        private int dia, mes, ano ; // tres atributos inteiros
        private static int [ ] numDias  = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        public Date4( ) { // construtor default sem parâmetros inicilaizdos com valores validos
                dia = 1 ; mes = 1 ; ano = 1900;
        }

        public Date4 ( int dia , int mes, int ano ) {// construtor com parâmetros passados por valor
                if ((mes > 0) && (mes <= 12)) this.mes = mes;
                else this.mes = 1 ;
                if (( dia > 0) && (dia <= numDias[mes - 1])) this.dia = dia ;
                else this.dia = 1 ;
                if ( ano > 1900) this.ano = ano ;
                else this.ano = 1900;
        }


        public bool anoBisexto ( ) {
                if ( ano % 4 != 0)
                        return false ;
                else if ( ano % 100 != 0)
                        return true ;
                else if ( ano % 400 != 0)
                        return false ;
                else return true ;
        }

        public int getDia ( ) { return dia ; }
        public int getMes ( ) { return mes ; }
        public int getAno ( ) { return ano ; }

}


class NovaData : Date4 {
        private static String [ ] NomeMeses = {"Janeiro", "Fevereiro", "Março",
        "Abril", "Maio", "Junho", "Julho ", "Agosto" ,
        "Setembro", "Outubro" , "Novembro" , "Dezembro" } ;
        public String toString ( ) {
                String temp = getDia ( ) + " de " + NomeMeses [getMes() - 1]  + " de " + getAno ( ) ;
                return temp ;
```

```
        }


}

class Program {
        static void Main(string[ ] args){
                NovaData d1 = new NovaData ( ) ;
                Console.WriteLine( "\nData padrao: " + d1.getDia ( ) + "/" +
                d1.getMes ( ) + "/" + d1.getAno ( ) ) ;
                Console.WriteLine( "Data padrao: " + d1.toString ( ) );
                if (d1.anoBisexto ( ))
                        Console.WriteLine( "Ano eh Bi sexto " ) ;
                else
                        Console.WriteLine( "Ano nao eh Bisexto " ) ;
        }
}

}
```

5. EXERCÍCIOS PARA ENTREGAR

5.1 Utilizando obrigatoriamente o conceito de **herança,** fazer **programas** (codificado inicialmente em **Java** e depois em **C#**) que atenda os seguintes requisitos **(Exercício para entregar):**
Criar uma classe **base Telefone** e uma classe **TelefoneEletronico** derivada de **Telefone.** Em **Telefone,** crie um membro ***protected*** *TipoDoTelefone* do tipo string, e um método ***public*** *Ring( )* que imprime uma mensagem como: "Tocando o <TipoDoTelefone>." Em *TelefoneEletronico*, o construtor deve ajustar (***set***) o TipoDoTelefone para "Digital". No método *Run( )*, chamar o método *Ring( )* no *TelefoneEletronico* para testar a herança.

5.2 Utilizando obrigatoriamente o conceito de **herança,** fazer **programas** (codificado inicialmente em **Java** e depois em **C#**) que atenda os seguintes requisitos **(Exercício para entregar):**

Uma loja comercial tem 2 tipos de funcionários: vendedores e administrativos. Para todos a empresa precisa ter o registro do nome e RG do funcionário. Os vendedores têm um salário base, mas ganham também comissão de suas vendas. Os administrativos têm um salário fixo, mas podem ganhar horas extras adicionais.

Os vendedores devem ter um método que acumule o total de vendas durante o mês e um método que imprima seu salário total considerando que a comissão é de 5%. Para os administrativos as horas extras é que são acumuladas e pagas com o valor de um centésimo do salário por hora. Nos dois casos, o método que imprime o salário a receber zera os valores acumulados.

6. E/S: FLUXOS E ARQUIVOS

A entrada e saída em um sistema computacional se refere a qualquer troca de informação entre uma aplicação e o mundo exterior.

A leitura do que o usuário digita no teclado, o conteúdo obtido de um arquivo ou os dados recebidos pela rede são exemplos de entrada de dados. A impressão de mensagens no console, a escrita de texto em um arquivo ou envio de dados pela rede são exemplos de saída de dados.

A plataforma .NET oferece diversas classes e interfaces para facilitar o processo de entrada e saída de dados.

A E/S (entrada/saída) de **arquivos e fluxos** refere-se à **transferência de dados** de ou para uma mídia de armazenamento

Um **fluxo** é uma **abstraçã**o de uma **sequência de bytes** que podem ser usados para **ler e gravar** em dispositivos de E/S (entrada/saída) para armazenamento (disco), pipes de comunicação entre processos, memória ou um soquete de rede TCP/IP.

Arquivos são **mecanismos de abstração** que fornecem uma forma de armazenar e recuperar informações na hierarquia da memória (primária, secundária (discos magnéticos, óticos, SSDs, etc.) de um computador.

Na maioria das aplicações o arquivo é o elemento central. Os usuários desejam acessar, salvar e manter a integridade dos arquivos.

Um arquivo é um **conjunto de registros lógicos que são** mapeados pelo sistema operacional (SO) para um **conjunto de registros físicos.** São geralmente **armazenados em blocos**, sendo o sistema operacional responsável pela alocação destes blocos.

Um **arquivo** geralmente pode ser considerado uma **coleção ordenada e nomeada de bytes** com armazenamento **geralmente persistente ou não volátil**. São adequados para armazenar informação a longo prazo, grande quantidade de informação e para compartilhamento de informações. Podem armazenar caracteres, dados numéricos e dados binários como programas executáveis, imagens, etc.

Framework class library (FCL) plataforma .Net é uma biblioteca de tipos de classes, interfaces e valor que fornecem acesso à funcionalidade do sistema. É a base sobre a qual aplicações Framework, componentes e controles são construídos.

Todo o código gerenciado na plataforma .Net é organizado em grupos lógicos denominados classes. Estas classes são agrupadas em hierarquias chamadas “namespace”. As bibliotecas contidas no .Net framework estão contidas no namespace “System”.

O namespace **Sistema.IO** do FCL do C# contém várias classes que permitem leitura e gravação a arquivos e fluxos de dados, entre elas pode-se citar:

6.1 Fluxos em C#

O C# utiliza o **Stream** para **ler e escrever em fluxos (ou arquivos).** Sempre que uma aplicação lê ou escreve em fluxos (arquivos ou se liga a outro computador em uma rede), envia e recebe bytes. Assim, sempre que se pretender ler ou escrever dados (bytes) em fluxos (ou arquivos), deve ser utilizado ou criado um objeto Stream.

A **classe base Stream** e suas classes derivadas oferecem suporte para leitura e gravação de fluxo de bytes. Esta classe (**Stream**) e suas derivadas proporcionam uma visão geral dos diferentes tipos de entrada e saída, isolando o programador de detalhes específicos do sistema operacional e dos dispositivos subjacentes.

Fluxos envolvem três **operações fundamentais**:

- **Leitura**: transferência de dados de um fluxo para uma estrutura de dados, como uma matriz de bytes.
- **Gravação**: transferência de dados para um fluxo a partir de uma fonte de dados.
- **Busca:** consulta e modificação da posição atual em um fluxo.

Algumas classes de fluxo mais comumente usadas são:

- **FileStream**: leitura e gravação em um arquivo.
- IsolatedStorageFileStream: leitura e gravação em um arquivo no armazenamento isolado.
- MemoryStream:leitura e gravação na memória como o repositório de backup.
- BufferedStream: para melhorar o desempenho das operações de leitura e gravação.
- NetworkStream: para leitura e gravação via soquetes de rede.
- PipeStream: para leitura e gravação sobre pipes anônimos e nomeados.
- CryptoStream: para vincular fluxos de dados a transformações criptográficas.

Dependendo do repositório ou da fonte de dados subjacente, os fluxos podem oferecer suporte somente algumas dessas capacidades. Por exemplo, a classe **PipeStream** não oferece suporte à operação de busca. As propriedades **CanRead, CanWrite e CanSeek** de um fluxo especificam as operações às quais o fluxo oferece suporte.

6.2 Arquivos em C#

O **sistema de arquivos** é o conjunto de rotinas do sistema operacional que fornece serviços para os usuários e suas aplicações.

O sistema de gerenciamento de arquivos é uma das partes mais importantes do sistema operacional de um computador e trata como os arquivos são constituídos, estruturados, nomeados, acessados, utilizados, protegidos e implementados.

Para se trabalhar com arquivos é necessário também tratar com nomes, localização, diretórios, armazenamento em disco, etc.

A interface do usuário (máquina virtual) deve ser feita de uma forma que mantenha o usuário isolado dos detalhes a respeito de como as informações são armazenadas nos arquivos, e de como os discos efetivamente trabalham.

De modo semelhante ao que acontece com o gerenciamento de memória, devem existir mecanismos para alocar espaço no disco para abrigar um arquivo.

6.2.1 Informações sobre arquivos, diretórios (pastas) e unidades

Como já foi mencionado, o namespace **Sistema.IO** do FCL do C# contém várias classes que permitem acessar informações do sistema de arquivo, leitura e gravação em arquivos e fluxos de dados.

No.NET Framework, pode-se acessar informações do sistema de arquivo, usando as seguintes classes:

- System.IO.FileInfo
- System.IO.DirectoryInfo
- System.IO.DriveInfo
- System.IO.Directory
- System.IO.File

O **FileInfo** e **DirectoryInfo** classes que representam um arquivo ou diretório e conter propriedades que expõem a muitos dos atributos de arquivo suportados pelo sistema de arquivos NTFS. Elas também contêm métodos para abrir, fechar, mover e excluir arquivos e pastas.

É possível obter nomes de arquivos, pastas ou unidades por meio das chamadas:

DirectoryInfo.GetDirectories, DirectoryInfo.GetFiles e DriveInfo.RootDirectory.

É possível criar instâncias dessas classes, passando uma sequência de caracteres que representa o nome do arquivo, pasta ou unidade para o construtor:

```
System.IO.DriveInfo di = new System.IO.DriveInfo (@"C:\");
```

O System.IO.Directory e System.IO.File classes fornecem métodos estáticos para obter informações sobre arquivos e diretórios.

Directory tem métodos estáticos para criar, mover e enumerar pastas e subpastas.

File fornece métodos estáticos para a criação, cópia, exclusão, movimentação e abertura de arquivos com a ajuda na criação de objetos FileStream.

Alguns métodos mais comuns para arquivos das classes File e FileInfo

Tarefa	Métodos
Criar um arquivo	File.CreateText e File.Create
Anexar texto em um arquivo	File.AppendText
Renomear ou mover um arquivo	File.Move
Excluir um arquivo	File.Delete
Copiar um arquivo	File.Copy
Obter o tamanho de um arquivo	Propriedade FileInfo.Length
Obter os atributos de um arquivo	File.GetAttributes
Definir os atributos de um arquivo	File.SetAttributes
Determinar se um arquivo existe	File.Exists

Alguns métodos mais comuns para diretório das classes Directory e DirectoryInfo

Tarefa	Métodos
Criar diretório	Método Directory.CreateDirectory
Criar subdiretório	Método CreateSubdirectory
Renomear ou mover um diretório	Método Directory.Move
Excluir diretório	Método Directory.Delete
Descobrir o tamanho de um diretório	Classe System.IO.Directory
Determinar se um diretório existe	Método Directory.Exists

O exemplo 1 mostra vários modos de acessar informações sobre arquivos e pastas.

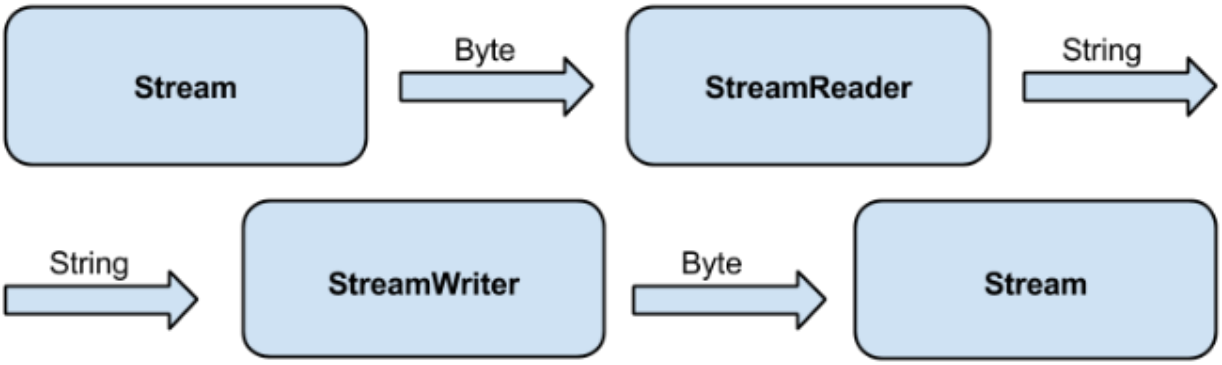
Exemplo1 (prática):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
// FileInfo.cs
namespace FileSysInfo {
  class FileSysInfo { {// arquivo fonte disponível no SGA
    static void Main( )    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");

        foreach (System.IO.FileInfo fi in fileNames){
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime,
               fi.Length);
        }

        // Get the subdirectories directly that is under the root.
        System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories("*.*");

        foreach (System.IO.DirectoryInfo d in dirInfos){
            Console.WriteLine(d.Name);
        }
        // The Directory and File classes provide several static methods
        // for accessing files and directories.
        // Get the current application directory.
        string currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Get an array of file names as strings rather than FileInfo objects.
        // Use this method when storage space is an issue, and when you might
        // hold on to the file name reference for a while before you try to
        // access the file.
        string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");
        foreach (string s in files)        {
            // Create the FileInfo object only when needed to ensure
            // the information is as current as possible.
            System.IO.FileInfo fi = null;
```

```
            try
            {
                 fi = new System.IO.FileInfo(s);
            }
            catch (System.IO.FileNotFoundException e)
            {
                // To inform the user and continue is
                // sufficient for this demonstration.
                // Your application may require different behavior.
                Console.WriteLine(e.Message);
                continue;
            }
            Console.WriteLine("{0} : {1}",fi.Name, fi.Directory);
        }

        // Change the directory. In this case, first check to see
        // whether it already exists, and create it if it does not.
        // If this is not appropriate for your application, you can
        // handle the System.IO.IOException that will be raised if the
        // directory cannot be found.
        if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\"))
        {
            System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");
        }

        System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

        currentDirName = System.IO.Directory.GetCurrentDirectory();
        Console.WriteLine(currentDirName);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
  }
}
```

6.3 LEITURA E ESCRITA DE ARQUIVOS NO C#

A E/S (I/O) do C# pode ser esquematizado pela seguinte figura:



A classe Stream está disponível na biblioteca IO, por isso deve-se incluir no arquivo da classe:

Using System.IO;

A entrada ou saída de dados com **StreamReader** no C# funciona em duas etapas.

Na primeira etapa, tem-se uma **classe abstrata** que representa uma sequência de bytes na qual pode-se realizar operações de leitura e escrita. Essa classe abstrata é chamada de Stream.

Como **Stream** é uma classe abstrata, não pode ser usada diretamente, é necessária uma implementação para essa classe. No caso de leitura ou escrita em arquivos, utiliza-se um tipo de Stream chamado FileStream, que pode ser obtido através do método estático Open da classe File.

Quando se utiliza Open, deve-se passar o nome do arquivo que será aberto e deve-se informá-lo o que se quer fazer com o arquivo (ler ou escrever).

Para abrir um arquivo entrada.txt para leitura, utiliza-se o código a seguir:

```
Stream entrada = File.Open("entrada.txt", FileMode.Open);
```

Agora que tem-se o Stream, pode-se ler seu próximo byte utilizando o métodoReadByte.

```
byte b = entrada.ReadByte();
```

A classe **Stream** foi criada para leitura e gravação de bytes, no entanto, trabalhar com bytes pode não ser conveniente e geralmente é melhor trabalhar com strings ou textos. Para facilitar a leitura e escrita de Streams, C# oferece as classes **StreamReader StreamWriter** para ler ou escrever as linhas de um arquivo de texto padrão, ou seja, podem ler caracteres ou strings de um Stream.

O StreamReader precisa saber qual é a Stream que será lida, portanto pode-se passar essa informação através de seu construtor:

```
StreamReader leitor = new StreamReader(entrada);
```

Para ler uma linha do arquivo, utiliza-se o método ReadLine do StreamReader:

```
string linha = leitor.ReadLine();
```

Enquanto o arquivo não terminar, o método ReadLine() devolve um valor diferente de nulo, deste modo, pode-se ler todas as linhas de um arquivo com o seguinte código:

```
string linha = leitor.ReadLine();
while(linha != null)  {
   Console.WriteLine( linha);
   linha = leitor.ReadLine();
}
```

Assim que terminar de trabalhar com o arquivo, deve-se sempre lembrar de fechar o Stream e o StreamReader:

```
leitor.Close();
entrada.Close();
```

O código completo (File1.cs) para ler de um arquivo fica da seguinte forma:

Exemplo2 (prática):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
// File1.cs
namespace File1 { // arquivo fonte disponível no SGA
  class File1 {
    static void Main()    {
          Stream entrada = File.Open ("entrada.txt", FileMode.Open);
          StreamReader leitor = new StreamReader(entrada);
          string linha = leitor.ReadLine();
          while(linha != null) {
                Console.WriteLine( linha);
                linha = leitor.ReadLine();
          }
```

```
            leitor.Close();
            entrada.Close();
        }
    }
}
```

Quando é passado apenas o nome do arquivo no código do File.Open, o C# procura esse arquivo dentro da pasta em que a aplicação é executada. No caso de executar a aplicação, a pasta utilizada pela aplicação será a pasta em que o projeto foi criado ou na pasta corrente (atual ou de trabalho) do console.

Porém, o arquivo pode não existir e, nesse caso, o C# lança a FileNotFoundException. Pode-se verificar se o arquivo existe antes de abri-lo utilizando se o método Exists da classe File:

```
if(File.Exists("entrada.txt"))
{
   // Aqui se tem certeza que o arquivo existe
}
```

O código da leitura com a verificação fica assim:

Exemplo3 (prática):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
// File2.cs
namespace File2 { // arquivo fonte disponível no SGA
  class File2 {
            static void Main( ){
                        if(File.Exists("entrada.txt")){
                                    Stream entrada = File.Open("entrada.txt", FileMode.Open);
                                    StreamReader leitor = new StreamReader(entrada);
                                    string linha = leitor.ReadLine();
                                    while(linha != null) {
                                                Console.WriteLine( linha);
                                                linha = leitor.ReadLine();
                                    }
                                    leitor.Close();
                                    entrada.Close();
                        }
            }
  }
}
```

Foi mostrado que para ler todas as linhas de um arquivo, pode-se utilizar o método ReadLine até que o retorno seja o valor null.

Ao invés de se chamar o método ReadLine para cada linha, pode-se utilizar o métodoReadToEnd da classe StreamReader. Esse método devolve uma string com todo o conteúdo do arquivo.

```
const string nomeArquivo = "banco.txt"; //nome do arquivo de dados
StreamReader arqDados = new StreamReader(nomeArquivot);    //abrir o arquivo
ou
StreamReader arqDados = new StreamReader(nomeArquivo, Encoding.Default);    //abrir o arquivo
```

OBS: o arquivo "banco.txt" deve estar no diretório de trabalho ou deve ser especificado seu caminho (PATH) completo.

Depois de associar um StreamReader a uma fonte de dados, pode-se fazer a leitura através do método **ReadLine**. Esse método devolve null quando o texto acaba

```
linha = arqDados.ReadLine();    //lê a 1a linha
while (linha != null) {          //enquanto houver dados...
   System.Console.WriteLine (linha); // imprime linha lida
    cont++;     //incrementa o contador de contas
    linha = arqDados.ReadLine();        //lê próxima linha
 }
 arqDados.Close();       //fecha arquivo
```

Exemplo4 (prática):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
// File3.cs
namespace File3 {
  class File3 { // arquivo fonte disponível no SGA
          static void Main( ){
                    string linha;
                    int cont = 0;
                    const string nomeArquivo = "banco.txt"; //nome do arquivo de dados
                    StreamReader arqDados = new StreamReader(nomeArquivo);    //abrir o arquivo
          // ou StreamReader arqDados = new StreamReader(nomeArquivo, Encoding.Default);
//abrir o arquivo
                    linha = arqDados.ReadLine();    //lê a 1a linha
                    while (linha != null) {          //enquanto houver dados...
```

```
                System.Console.WriteLine (linha); // imprime linha lida
                cont++;     //incrementa o contador de contas
                linha = arqDados.ReadLine();        //lê próxima linha
            }
            System.Console.WriteLine ("Numero contas lidas: " + cont); // imprime linha lida
            arqDados.Close();       //fecha arquivo
        }
    }
}
```

6.3.2 LEITURA DE ARQUIVOS TEXTO COM TextReader

TextReader e TextWriter são usadas para ler ou escrever uma série sequencial de caracteres.

Para ler um texto de um arquivo do teclado ou de qualquer outra fonte de dados, pode-se criar objetos da classe **System.IO.TextReader** e associá-los a uma determinada fonte de dados (teclado, arquivo, rede, entre outros). **TextReader** representa um leitor que pode ler uma série sequencial de caracteres.

Exemplo5 (prática):

```
TextReader leitor = new StreamReader ("entrada.txt"); //  Criar um TextReader associado a um arquivo, sendo que  o arquivo "entrada.txt" deve estar no diretório de trabalho ou deve ser especificado seu caminho (PATH) completo.
```

Exemplo6 (prática):

```
TextReader leitor = System.Console.In; // Criar um TextReader associado ao teclado
```

Depois de associar um TextReader a uma fonte de dados, pode-se fazer a leitura através do método **ReadLine**. Esse método devolve null quando o texto acaba

```
TextReader leitor = ...

string linha = leitor . ReadLine (); //lê a 1a linha
while ( linha != null ) { //enquanto houver dados.
        System . Console . WriteLine (linha); // imprime linha lida
linha = leitor.ReadLine ( ); //lê próxima linha

 }
```

6.3.3 ESCRITA DE ARQUIVOS TEXTO com StreamWriter

Assim como a leitura, a escrita também acontece em duas etapas. Na primeira etapa, escreve-se bytes para a saída. Para isso será utilizada novamente a classe abstrata Stream.

Para escrever em um arquivo, primeiro deve-se abri-lo no modo de escrita utilizando-se o método Open do File passando o modo FileMode.Create:

```
Stream saida = File.Open("saida.txt", FileMode.Create);
```

Para não trabalhar com bytes, pode-se utilizar a classe StreamWriter para em escrever em Stream.

```
StreamWriter escritor = new StreamWriter(saida);
```

Pode-se escrever uma linha com o StreamWriter utilizando o método WriteLine:

```
escritor.WriteLine("minha mensagem");
```

Depois que terminar de utilizar o arquivo, é preciso fechar todos os recursos:

```
escritor.Close();
saida.Close();
```

O código completo para escrever no arquivo fica da seguinte forma:

Exemplo7 (prática):

```
Stream saida = File.Open("saida.txt", FileMode.Create);
StreamWriter escritor = new StreamWriter(saida);
escritor.WriteLine("minha mensagem");
escritor.Close();
saida.Close();
```

Observe que, por usar uma classe abstrata, pode-se então trocar facilmente a classe concreta por outra. Por exemplo, poderia ler de um Socket, ou de uma porta serial, e o código seria o mesmo: basta a classe ser filha de Stream. Repare que o uso de classes abstratas e polimorfismo nos possibilita ler/escrever em diferentes lugares com o mesmo código.

Exemplo8 (prática):

```
StreamWriter arqSaida = new StreamWriter(nomeArquivo, false, Encoding.Default); //abre arquivo para escrita
ou
StreamWriter arqSaida = new StreamWriter(nomeArquivo); //abre arquivo para escrita
```

Depois de associar um **StreamWriter** a um destino de dados, pode-se fazer a escrita através do método *WriteLine*. para arquivos, é importante fechar o StreamWriter após escrever o conteúdo.

```
StreamWriter arqSaida = new StreamWriter(nomeArquivo); //abre arquivo
String linha;          //para formatar a linha
 for ( i = 0; i < num_linhas; i++) {    //percorre todo o vetor
          aux = vetContas[i];
           linha = aux.agencia + ";" + aux.numero + ";" +  aux.obterSaldo();
            //monta a linha a ser gravada, usando ; como separador
            arqSaida.WriteLine(linha);      //escreve no arquivo
}
arqSaida.Close();  //fecha o arquivo
```

Exemplo9 (prática):

Para criar um objeto do tipo writer pode-se fazer:

StreamWriter wr = new StreamWriter(@"c:\pasta\arquivo.txt", true);

OBS:
Colocar o @ antes do path do arquivo faz com que o compilador não interprete a barra "\" como sendo uma mudança de linha ou tabulação (\n ou \t), e sim como uma string.

A seguir pode-se escrever no arquivo a função de escrita:

wr.WriteLine("Este é o texto a escrever no arquivo");

Não esquecer de fechar o arquivo no final da escrita para que não fique aberto para outros programas:

wr.Close();

Exemplo10 (prática):

Leitura de arquivos com o StreamReader e escrita com o StreamWriter:

StreamReader rd = new StreamReader(@"c:\pasta\ficheiro.txt");

Para melhor exemplificar, será criado um arquivo onde se escreverá duas linhas e, depois de escrito, vai ler essas duas linhas.

```
StreamWriter wr = new StreamWriter(@"c:\pasta\doc.txt", true);
wr.WriteLine("Primeira linha");
wr.WriteLine("Segunda linha");
wr.Close();

StreamReader rd = new StreamReader(@"c:\pasta\doc.txt");
while(!rd.EndOfStream){
        String linha = rd.ReadLine();
        Console.WriteLine(linha);
}
rd.Close();
```

No ciclo while, são lidas todas a linhas enquanto estas tiverem algo escrito (EndOfStream).

Tal como na escrita, também na leitura deve-se fechar o arquivo com Close().

Exemplo11 (prática):
Programa de teste para recuperar e imprimir na tela o conteúdo digitado pelo usuário no teclado.

OBS: Para finalizar o fluxo de entrada do teclado digite CTRL+Z.

```
using System ;
using System .IO;
public class LeituraDoTeclado{
        static void Main  ( ) {
                TextReader teclado = Console .In;
                string linha = teclado . ReadLine ();
                while ( linha != null ) {
                        System . Console . WriteLine ( linha );
                        linha = teclado . ReadLine ();
                }
        }
}
```

Exemplo12 (prática):
Programa de teste para recuperar e imprimir na tela o conteúdo de um arquivo.

```
using System ;
using System .IO;
public class LeituraDeArquivo{
        static void Main  ( ) {
                TextReader arquivo = new StreamReader (" entrada . txt ");
                String linha = arquivo . ReadLine ();
                while ( linha != null ) {
                        System . Console . WriteLine ( linha );
                        linha = arquivo . ReadLine ();
                }
                arquivo . Close ();
        }
}
```

OBS: O arquivo “entrada.txt” deve ser criado no diretório de trabalho ou deve ser especificado seu caminho (PATH) completo.

Exemplo13 (prática):
Programa de teste para imprimir algumas linhas em um arquivo.

```
using System ;
using System .IO;
public class EscritaDeArquivo {
        static void Main ( )  {
                TextWriter arquivo = new StreamWriter (" saida . txt ");
                arquivo .WriteLine (" Primeira linha !!! ");
                arquivo . WriteLine (" Segunda linha !!! ");
                arquivo . WriteLine (" Terceira linha !!! ");
                arquivo . Close ();
        }
}
```

6.3.4 ESCRITA DE ARQUIVOS TEXTO COM TextWriter

Para escrever um texto em um arquivo, na tela ou de qualquer outro destino de dados, pode-se criar objetos da classe **System.IO.TextWriter** e associá-los a um determinado destino de dados (tela, arquivo, rede, entre outros).

Exemplo14 (prática):

```
TextWriter escritor = new StreamWriter ("saida. txt "); // Criar um TextWriter associado a um arquivo
ou

TextWriter escritor = System.Console.Out ;  // Criar um TextWriter associado a tela
```

Depois de associar um TextWirter a um destino de dados, pode-se fazer a escrita através do método *WriteLine*. para arquivos, é importante fechar o TextWriter após escrever o conteúdo.

```
TextWriter escritor = ...
escritor.WriteLine ("oi");
escritor.WriteLine ("oi oi");
escritor.WriteLine ("oi oi oi");
escritor.WriteLine ("oi oi oi oi");
escritor.Close ( );  // fecha arquivo
```

OBS:
Ver no SGA o programa exemplo conta_arq_sga.cs).
Pode-se usar também **StreamWriter** (ver no SGA o programa exemplo conta_arq_sga.cs).

7. EXERCÍCIOS PARA ENTREGAR (Exercícios para entregar)

7.1 Fazer **programas** inicialmente codificado em **C#** que implemente a **classe Pessoa,** com as seguintes especificações **(Exercício para entregar)**:
7.1.1 Atributos: data de nascimento, peso e altura.
7.1.2 Métodos de acesso aos atributos (get / set);
7.1.3 Um construtor que receba valores para todos os atributos da classe.
7.1.4 Um método informar a idade atual da pessoa;
7.1.5. Um método para calcular o IMC. Para calcular o IMC, deve-se dividir o peso (em kg) da pessoa pela altura (em metros) ao quadrado.
7.1.6 O programa deve ler os dados de pessoas a partir de um arquivo texto. Nesse arquivo, cada linha contém a data de nascimento, o peso e a altura de uma pessoa, sendo esses dados separados por ":".

7.1.7 O programa também deve ter a opção de inserir dados de uma nova pessoa no sistema.
7.1.8 Antes do programa ser finalizado, deve atualizar o arquivo de dados com as novas pessoas cadastradas.
7.1.9 Usar as técnicas de encapsulamento e ocultação explicadas nas aulas teóricas

7.2 A partir da classe **Conta** (implementada nos exercícios anteriores) **e utilizando obrigatoriamente** o conceito de **herança,** fazer um **programa** codificado em **C#** que implemente as **classes Poupança e Investimento,** com as seguintes especificações **(Exercício para entregar):**
7.2.1 Atributos adicionais obrigatórios:

```
Date abertura; // data de abertura da conta
private int tipoConta;   // indica o tipo da conta:  corrente, poupança ou investimento
private double saldo;   // saldo atual da conta
private double taxa_juros;   // taxa de juros mensal da poupança
private double rendimento;   // taxa de rendimento mensal do investimento
private double imposto;   // taxa de imposto mensal do investimento
```

7.2.2 Métodos adicionais obrigatórios:

```
public double obterSaldo_poupanca(int numConta);
public void depositar(double credito, int numConta);
public void sacar(double quantia, int numConta);
public void ImprimeSaldo (int numConta);
}
```

7.2.3 Os dados das contas devem ser armazenados em vetor estático do tipo abstrato Conta com limite de 100 clientes, ou seja:

```
const int MAXCONTAS = 100; // número máximo de contas suportado
static Conta[ ] vetContas = new Conta[MAXCONTAS]; //vetor de contas
```

7.2.4 O programa deve apresentar inicialmente na tela um menu com as seguintes opções:

1. Criar uma conta nova.
2. Excluir uma conta existente
3. Depositar em uma conta
4. Sacar de em uma conta
5. Imprimir saldo de uma conta
6. Imprimir uma relação das contas existentes informando o número da conta e o nome do titular da conta
7. Sair do programa

7.2.5 O programa deve obter a opção do usuário, chamar o método correspondente, apresentar o resultado e sempre voltar ao menu inicial, exceto quando for selecionada a opção 7 (Sair do programa).

7.2.6 No saldo das contas de poupança deve ser considerado o rendimento mensal obtido com a taxa de juros mensal da poupança. Neste tipo de conta não há tributação.

7.2.7 No saldo das contas de investimento deve ser considerado o rendimento mensal do investimento, abatendo-se o imposto sobre o rendimento bruto obtido.

7.2.8 Os **dados dos clientes** devem estar **armazenados em arquivos**, sendo cada agência deve ter um arquivo exclusivo, ou seja, os dados da agência 1 ficam armazenados no arquivo agencia1.txt, os dados da agência 2 ficam armazenados no arquivo agencia2.txt e assim por diante.

```
//Arquivo exemplo (nome; agência, conta, tipo da conta, saldo bruto):
//Joao Sousa;0001;001;1;2500,00
//Maria Sampaio;0001;013;3;4585,33
```

7.2.9 O programa deve trabalhar com **uma agência de cada vez** (agência de atual) e ao ser iniciado deve solicitar o número da agência e no caso da agência não estiver cadastrada, deve oferecer a possiblidade de cadastrar uma nova agência.

7.2.10 O programa deve permitir a troca da agência de atual e neste caso deve salvar os dados da agência atual antes de efetuar a troca de agência.

7.2.11 Para testar, considere nas contas de poupança uma taxa de juros de 0,5% ao mês, nas contas de investimento um taxa de rendimento mensal do investimento de 0,65% ao mês e o imposto de 15% sobre o rendimento bruto obtido.

7.1.12 Usar as técnicas de encapsulamento e ocultação explicadas nas aulas teóricas

7.2.13 Outras especificações adicionais podem ser fornecidas pelo professor durante as aulas