3. PRIMEIROS PROGRAMAS EM JAVA

A seguir, será apresentado , na prática, como os programas Java podem ser criados, compilados e executados.

3.1. Criando o Primeiro Programa em Java

1. Escrever o código em um arquivo texto com um editor;

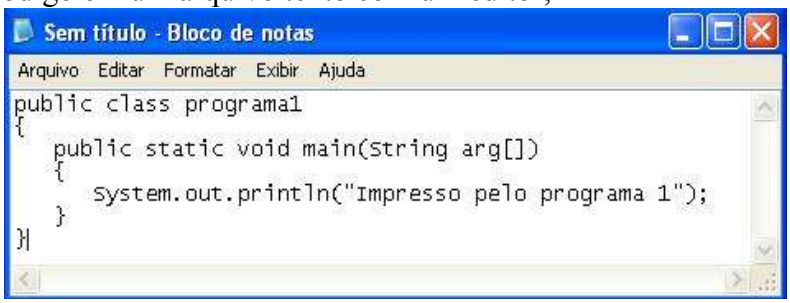


Figura 10 - Primeiro Programa em Java

2) Salvar o arquivo com o nome "**programa1.java**" em uma pasta. **EX: c:\java;**

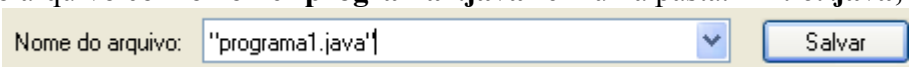


Figura 11 - Salvar o arquivo .java

3. Pelo prompt do DOS:
3.1. Compilar o código: **javac programa1.java** (gera o programa1.class)
3.2. Executar o aplicativo: **java programa1** (sem o .class)

Figura 12 - Compilando e Executando

3.2. Programa Bem Vindo

```
public class BemVindo {
   public static void main( String args[] )  {
      System.out.println( "Bem vindo ao " );
      System.out.println( "mundo Java!" );
   }
}
```

3.3. Programa Bem Vindo 2

```
public class BemVindo2 {
   public static void main( String args[] )  {
      System.out.println( "Bem vindo\nao\nmundo\nJava!" );
   }
}
```

5. SINTAXE DO JAVA

5.1. Variáveis

No Java, uma variável pode ter basicamente três tipos.

- **Primitivas**: (valores atômicos) variáveis de "baixo nível", como números inteiros, com casas decimais, caracteres e boleanas (sim/ não, verdadeiro / falso);
- **Compostas / Por Referência**: representam uma estrutura de dado já preparada para um determinado fim como Data, Conta, Produto, Moeda, String / Cadeia de caracteres, etc.;
- **Arrays**: representam listas que podem assumir qualquer tipo primitivo ou composto.

As variáveis tem seu acesso limitado de acordo com o seu escopo.
A sintaxe para declaração de uma variável é simples:

modificadores tipo nome;

A seguir será demonstrado um exemplo de declaração de variável:

private double m ;

O modificador private, especifica o acesso para a variável, nesse caso, a variável será acessível somente no local em que foi declarada. A palavra double representa o tipo da variável e m é o nome dado a variável.

5.1.1. Variáveis de Tipo Primitivo

As variáveis devem possuir nome, tipo e valor, sendo que toda vez que se necessite usar uma variável no Java, é preciso declará-la, para então atribuir valores a mesma. A seguir são apresentados os tipos primitivos usados para definir variáveis na linguagem Java:

Tipo	Sorte	Valor mínimo	Valor máximo	Valor inicial default
boolean	Valores lógicos	true	false	false
byte	Inteiros de 8 bits	-128	+127	0
short	Inteiros de 16 bits	-32.768	+32.767	0
char	Caracteres codificados em Unicode de 16 bits	\u0000	\uFFFF	\u0000
int	Inteiros de 32 bits	-2.147.483.648	+2.147.483.647	0
long	Inteiros de 64 bits	-9.223.372.036.854.775.808	+9.223.372.036.854.775.807	0
float	Pto. Flut. de 32 bits	NEGATIVE_INFINITY	POSITIVE_INFINITY	0
double	Pto. Flut. de 64 bits	NEGATIVE_INFINITY	POSITIVE_INFINITY	0

Figura 17 - Tipos primitivos do Java

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

         int numetoInteiro; // apenas declara a variável
         numetoInteiro = 5; // inicializa a variável

        double numeroDecimal = 2.34545; // declara e inicializa
        char letra = 'a'; // declarando e inicializando um char
        byte  a = 127;
        short b = 32767;
        long  c = 92233720368547758O7L; // L no final
        float  d = 1292.74F;  // F (ou f) após o literal indica precisão simples
        boolean e = true;

        // Definindo mais de uma variável ao mesmo tempo
        int x, y, z;

        // Definindo e inicializando mais de uma variável
        int q = 55, w = 6, t = 44;
    }
`
```

Figura 18 - Exemplos de Tipos Primitivos

5.1.1.1 Classes Wrapper

Cada classe de tipo primitivo possui uma classe wrapper para operações sobre cada tipo de dado. Segue a lista de classes wrapper para cada tipo primitivo:

int -> Integer	boolean -> Boolean	byte -> Byte
char -> Character	short -> Short	long -> Long
float -> Float	double -> Double	

Algumas operações comummente utilizadas:
Integer.parseInt(String s): Converte uma string para inteiro
Double.parseDouble(String s): Converte uma string paradouble
Float.parseFloat(String s): Converte uma string para float

5.1.2. Variáveis Compostas / Referência

Tipos que são definidos por uma classe. Exemplos: String, Date, List, Vector, etc.

```
public class TiposCompostos {
    public static void main(String[] args) {
        // Usando a classe String
        String palavra = "Teste";
        String frase;
        frase = "Teste de uma frase";
        String teste2 = new String("Inicializar pelo construtor da classe");

       // Usando a classe Date - é uma classe do pacote java.util
        // Pegando a data do Sistema Operacional
        java.util.Date hoje = new java.util.Date();

        // Inicializando uma data no construtor da classe
        java.util.Date dia = new java.util.Date("11/22/1974");

        // pode-se criar usando uma classe própria
        MinhaClasse objeto1 = new MinhaClasse();
        objeto1.meuMétodo();
    }
}
```

Figura 19 - Exemplos de Tipos Complexos

5.2. Declarando Variáveis

As declarações de variáveis consistem de um tipo e um nome de variável: como segue o exemplo:

```
int idade;
String nome;
boolean existe;
double salario;
```

Os nomes de variáveis podem começar com uma letra, um sublinhado (_), ou um cifrão ($). Elas não podem começar com um número. Depois do primeiro caracter pode-se colocar qualquer letra ou número.

5.3. Atribuições a variáveis

Após declarada uma variável a atribuição é feita simplesmente usando o operador '=':

```
idade = 18;
nome = "Fulado de Tal";
existe = true;
salario = 1550.50;
```

5.4. Conversoes de Tipos por Cast

Pode-se atribuir uma variável primitiva usando um valor literal ou o resultado de uma expressão. Números inteiros literais (como 4 por exemplo) sempre serão implicitamente um tipo int, no capitulo anterior foi visto que um tipo int é um valor de 32 bits. Não haverá problema em atribuirmos um valor a uma variável int ou long, mas o que aconteceria se atribuíssemos um valor inteiro a uma variável byte de 8 bits? Um byte não poderá armazenar tantos bits quanto uma variável do tipo int, mas então como o seguinte código é executado?

```
byte b = 30;
```

Este código somente funciona porque o compilador java compacta automaticamente o valor literal (30) para o tipo byte. Em outras palavras o compilador executa o cast. Também poderíamos moldar explicitamente o valor da seguinte forma:

```
byte b = (byte) 30
```

O exemplo acima é exatamente igual ao anterior, porem com uma modelagem (cast) explicita sobre o valor 30 que originalmente é um valor int de 32 bits.

O mesmo ocorre com valores de ponto flutuante porem, com algumas diferenças. Com estes o tipo padrão de armazenamento é o tipo double, qualquer numero de ponto flutuante atribuído a uma variável será considerado como um double a menos que seja explicitamente declarado no código que deseja-se utilizar outro tipo, como um float. Mas ao contrario do que acontece nos tipos inteiros um double não é automaticamente moldado para os demais tipos de dados de menor precisão. O código abaixo não será compilado:

```
float f = 34.5;
```

Poderíamos imaginar que o valor 34.5 caberia tranquilamente em uma variável com o tamanho de um float, mas por segurança o compilador não permitirá isso, justificado pela perca de precisão de tal transformação. Para atribuir um valor literal de ponto flutuante a uma variável do tipo float precisamos modelar o valor ou então acrescentar um f ao final do literal, Mas lembre-se de que com isso você estará truncando os bits mais a esquerda do valor, neste caso específico de um cast de double para float metade dos bits serão descartados. Os exemplos abaixo serão compilados e executados corretamente:

```
float f = (float) 34.5;
float f = 34.5f;
float f = 34.5F;
```

5.5. Comentários

Java possui três tipos de comentário:

1. /* ... */: como no C e C++, tudo que estiver entre os dois delimitadores são ignorados:

```
/* Este comentário ficará visível somente no código o compilador ignorará completamente este
trecho entre os delimitadores
*/
```

2. Duas barras (//): também podem ser usadas para se comentar uma linha:

```
      int idade; // este comando declara a variável idade
```

3. / ... */:** Este comentário é especial e é usado pelo **javadoc** para gerar uma documentação do código.

5.6. Caracteres especiais

Caracter	Significado
\n	Nova Linha
\t	Tab
\b	Backspace
\r	Retorno do Carro
\f	"Formfeed" (avança página na impressora)
\\	Barra invertida
\'	Apóstrofe
\"	Aspas
\ddd	Octal
\xdd	Hexadecimal

5.7. Operadores

5.7.1. Operadores Aritméticos

Operador	Significado	Exemplo
+	soma	3 + 4
-	subtração	5 - 7
*	multiplicação	5 * 5
/	divisão	14 / 7
%	módulo	20 % 7

Exemplo Aritmético:

```
class ArithmeticTest {
public static void main ( Strings args[] ) {
      short x = 6;
      int y = 4;
      float a = 12.5f;
      float b = 7f;

      System.out.println ("x é " + x + ", y é " + y );
      System.out.println ("x + y = " + (x + y) );
      System.out.println ("x – y = " + (x – y) );
      System.out.println ("x / y = " + (x / y) );
      System.out.println ("x % y = " + ( x % y ) );

      System.out.println ("a é " + a + ", b é " + b );
      System.out.println (" a / b = " + ( a / b ) );
}
}
```

A saída do programa acima é :

```
x é 6, y é 4
x + y = 10
x – y = 2
x / y = 1
x % y = 2
a é 12.5, b é 7
a / b = 1.78571
```

5.8. Mais sobre atribuições

Variáveis podem atribuídas em forma de expressões como:

```
int x, y, z;
x = y = z = 0;
```

No exemplo as três variáveis recebem o valor 0;

Operadores de Atribuição:

Expressão	Significado
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y

5.9. Incrementos e decrementos

Como no C e no C++ o Java também possui incrementadores e decrementadores:

y = x++;

y = - -x;

As duas expressões dão resultados diferentes, pois existe uma diferença entre prefixo e sufixo. Quando se usa os operadores (x++ ou x--), y recebe o valor de x antes de x ser incrementado, e usando o prefixo (++x ou --x) acontece o contrario, y recebe o valor incrementado de x.

5.10. Comparações

Java possui vários operadores para testar igualdade e magnitude. Todas as expressões retornam um valor booleano (true ou false).

5.11. Operadores de comparação

Operador	Significado	Exemplo
==	Igual	x == 3
!=	Diferente (Não igual)	x != 3
<	Menor que	x < 3
>	Maior que	x > 3
<=	Menor ou igual	x <= 3
>=	Maior ou igual	x >= 3

5.12. Operadores lógicos

Operador	Significado
&&	Operação lógica E (AND)
||	Operação lógica OU (OR)
!	Negação lógica

5.13. Operadores de Atribuição

Java oferece vários operadores de atribuição que abreviam as expressões de atribuição:

```
c = c + 3;        c += 3;          //    c = c * 3;  c *= 3;
c = c / 3;        c /= 3;          //    c = c - 3;  c -= 3;
c = c % 3;        c %= 3;
```

Java oferece operador de incremento unário, ++, e o operador de decremento unário, --.

```
                  Depois      Antes
c = c + 1;        c++;        ++c;
C = c - 1;        c--;        --c;
```

Exemplo:

```
int c = 1;
 System.out.println(c++);  // Saída 1
 System.out.println(c);      // Saída 2
```

5.14. Expressões

Expressões fazem combinações ordenadas de valores, variáveis, operadores, parênteses e chamadas de métodos, permitindo realizar cálculos aritméticos, concatenar strings, comparar valores, realizar operações lógicas e manipular objetos.

As expressão são avaliadas obedecendo as regras da Matemática. Um exemplo de regra a ser seguida é a da associatividade. Se um operador de mesmo nível aparece mais de uma vez em uma expressão, como em **a+b+c**, então o operador mais à esquerda é avaliado primeiro, sendo seguido pelo da direita, e assim por diante. A regra de associatividade pode ser visualizada no exemplo a seguir:

Exemplo 1: ((a+b)+c);

Em alguns casos é necessário utilizar operadores de diferentes níveis em uma mesma expressão fazendo com isso a aplicação da regra de precedência de operadores, como mostrado no exemplo a seguir:

Exemplo 2: a+b*c;

No exemplo acima multiplicou-se primeiro b com c e em seguida o somou-se o produto com o valor de a, pois a multiplicação e a divisão tem precedência sobre a adição e subtração.

5.15. Blocos de Código

Um bloco é definido por ({}) e contém um grupo de outros blocos. Quando um novo bloco é criado um novo escopo local é aberto e permite a definição de variáveis locais. As variáveis definidas dentro de um bloco só podem ser vistas internamente a este, e são terminadas ou extintas no final da execução do bloco.

```
void testabloco(){
      int x = 10, w=1;

      if (x> w)
      {  // inicio do bloco
            int y=50;
            System.out.println("dentro do bloco");
            System.out.println("x: " + x);
            System.out.println("y: " + y);
      } // final do bloco
      System.out.println("w: " + w);
      System.out.println("y: " + y); // erro variável não conhecida
}
```

5.16. Controle de Fluxo

As linguagens de programação oferecem controles de fluxo (condicionais e de repetição), para que os comandos possam ser executados em diferentes partes de um programa, baseado em condições definidas.

Os comandos condicionais em Java são classificados em categorias como pode ser visto abaixo:

Comando	**Palavras-chave**
Tomada de decisões	if-else, switch-case
Laços ou repetições	for, while, do-while
Apontamento e tratamento de excessões	try-catch-finally, throw

5.17. Tomada de Decisões

5.17.1. Estrutura de seleção if

O comando if-else permite escolher alternadamente entre dois outros comandos a executar. Se o valor da expressão condicional for true, então o primeiro bloco/comando será executado, do contrário, o segundo.

Sintaxe:

if (*expressão*)
  comando
[**else**
  comando]

Exemplo 1:

```
if (resposta == true) {
   // comandos
}
```

Exemplo 2:

```
if (resposta == true) {
     // comandos
}
else {
   //comandos
}
```

5.17.2. Estrutura de seleção switch

Semelhante ao comando if, no entanto, permite a seleção múltipla, podendo testar vários valores em uma expressão.

Sintaxe:

switch (*expressão*) {
  case *valor1*: *comando*; [**break**];
  case *valor2*: *comando*; [**break**];
  ....
  [**default:** *comando*;]
}

Exemplo 1:

```
int mês = 3;
switch (mes)
{
case 1: System.out.println("Jan");
       break;
case 2: System.out.println("Fev");
       break;
       ....
}
```

5.18. Estruturas de Repetição

5.18.1. while

O *while* é utilizado para repetir um comando, ou um conjunto de comandos, enquanto a condição for verdadeira.

Sintaxe:

while (*expressão*)
comando

Exemplo:

```
int i = 3;
while (i <= 10) {
       i++;
       System.out.println(i);
}
```

5.18.2. for

O laço de repetição *for* tem a função de executar um determinado número de vezes um determinado comando/bloco. A sintaxe do comando for pode ser vista abaixo.

Sintaxe:

for (*inicio*; condição; incremento)
comando

Exemplo:

```
for (int i=1; i < 10; i++)
       System.out.println(i);
```

5.18.3. do/while;

Este comando, não muito diferente do comando while, é um tipo de laço de repetição que executa um comando/bloco e em seguida avalia a expressão condicional.

Sintaxe:

do {
comandos
}
while (*expressão*);

Exemplo:

```
do {
 System.out.println("Indice:"+a);
 a++;
} while (a != 5);
```

5.19. Instruções break e continue;

Na execução de uma estrutura de repetição (for, while e do/while), a instrução **break** ocasiona a saída imediata da estrutura, enquanto a instrução **continue** pula as instruções restantes e prossegue com o teste para a próxima iteração do laço:

Exemplo de Continue:

```
for(int a=1; a<10; a++)
{
     if(a == 5)
          continue;
     saida += a + "  ";
}
```

Exemplo de Break:

```
a = 10;
somatorio = 0;
do {
     if(a == 0)
          break;
     somatorio += 10 / a--;
} while (true);
```

5.20. Instruções rotuladas

Para que o fluxo de execução continue ou pare, respeitando blocos definidos pelo programador, utilizamos as instruções rotuladas:

```
stop: {
        for(int i=0; i<5; i++)
           for(int j=0; j<3; j++) {
              if(i == 2)
                 break stop;
              saida += "["+i+" "+j+"] ";
           }
        }
```

5.21. Exercício 1 - Fatorial

Faça um programa em Java que solicite ao usuário um número que será utilizado para calcular o seu fatorial. Mostre o resultado ao usuário.

Utilize a estrutura de repetição for

Fatorial de 5:
5 * 4 * 3 * 2 * 1 = 120

5.22. Exercício 2 - Fibonacci

Fazer um programa que imprime a seguinte seqüência: 1, 1, 2, 3, 5, 8, 13, ... (serie de Fibonacci)

Utilize a estrutura de repetição while
Imprimir somente os valores menores que 100.

7. COVENÇÕES PARA NOMES DE CLASSES E VARIÁVEIS

Para a escrita de programas em Java, algumas convenções para declaração e escrita de classes e variáveis podem ser seguidas, para que assim mantenha-se um padrão de nomenclatura. Segundo Castela (2006), ”Como o Java é case-sensitive, todo e qualquer comando, variável, nome de classes e objetos escritos em Java será diferenciado entre maiúsculas e minúsculas.”, o Java impõem diferença de caixa alta de caixa baixa, como é ilustrado no exemplo a seguir:

Exemplo 1:

```
int Exemplo;
int exemplo;
int eXemplo;
```

Como mostra o exemplo acima, as variáveis são do mesmo tipo, porém diferenciam-se uma das outras, pela forma que forma que foram escritas. As variáveis em Java são declaradas com letras minúsculas, porém se haver uma palavra composta, declara-se a primeira como minúscula e a segunda com a primeira letra do segundo nome em maiúscula, como exemplo tem-se as duas situações a seguir:

Exemplo 2:

Variáveis	**Descrição**
int nome;	formato de declaração com apenas uma palavra, todas as letras minúsculas
int nomeComposto;	formato de declaração com duas palavras, com a inicial da segunda palavra em maiúsculo
int nomeCompostoDois;	formato de declaração com várias palavras, com o restante das outras palavras com inicial em maiúsculo

Assim como nas variáveis, para criar uma classe no Java aconselha-se que algumas convenções sejam postas em prática. Toda classe começa com letras maiúsculas, mas se existir uma segunda palavra ou mais, estas deverão também iniciar com primeira letra em maiúscula. O exemplo a seguir demonstra como criar uma classe de acordo com a convenção:

Exemplo 3:

Classe	**Descrição**

public class Pessoa{
 	
}
	Formato de declaração de Classe com uma palavra

public class PessoaFisica{
 	
}
	Formato de declaração de Classe com duas palavras

public class PessoaFisicaJuridica{
 	
}
	Formato de declaração de Classe com várias palavras

8. A CLASSE STRING

Em Java string é uma classe ao invés de um tipo de dado, no caso um objeto String. Uma string caracteriza-se por ser uma seqüência de caracteres, como palavras, frases e nomes. A classe String fica no pacote java.lang, que é importado automaticamente pela aplicação.

Exemplo da Criação de uma String:

```
String nome = new String("Paulo");
```

Objetos String são os únicos em Java que podem ser criados sem o operador new,, atribuindo diretamente um valor literal, pois seu esse objeto é tão usado que os implementadores de Java disponibilizaram uma forma mais simples de instanciar objetos String.

Outra forma de criar uma String:

```
String nome = "Paulo";
```

Mas quando um objeto String é criado dessa forma, nos bastidores o compilador está fazendo **String nome = new String("Paulo");**

8.1. Comprimento da String

O método length permite descobrir o número de caráteres contidos numa String, por exemplo:

```
String s = new String( "Olá!" );
int tamanho = s.length();
```

Resulta o valor 4 para o inteiro chamado tamanho.

8.2. Concatenação

Para concatenar duas Strings, pode-se utilizar o operador + ou o método concat. A concatenação resulta na criação de um novo objeto.

Exemplo:

```
String s1 = "Olá ";
String s2 = "Mundo!";
System.out.println( s1.concat( s2 ) );
System.out.println( s1 + s2 );
```

A saída gerada pelo exemplo acima será:

```
Olá Mundo!
Olá Mundo!
```

8.3. Comparação

Para determinar se duas Strings possuem o mesmo conteudo, usa-se o método **equals()**. Este método considera maiúsculas e minúsculas como diferentes. Para não fazer esta distinção, pode-se usar o método **equalsIgnoreCase()**.

É importante não confundir estas comparações de conteudo com o uso do operador = =, que, quando usado entre duas variáveis que são referências a objetos, serve para determinar se as duas variáveis apontam para o mesmo objeto.

Observe o exemplo abaixo:

```
String s1,s2,s3,s4,s5,s6;
boolean b1, b2, b3, b4, b5, b6, b7, b8, b9;

s1 = "Olá!";
s2 = "Olá!";
s3 = "olá!";
s4 = new String( "Olá!" );
s5 = s4;
s6 = new String( "Olá!" );

b1 = s1.equals( s2 );
b2 = s1.equals( s3 );
b3 = s1.equalsIgnoreCase( s3 );
b4 = s1.equals( s4 );
b5 = ( s1 == s2 );
b6 = ( s1 == s4 );
b7 = ( s4 == s5 );
b8 = ( s4 == s6 );
b9 = s4.equals( s6 );
```

Neste exemplo, vão ficar com o valor false somente b2, b6 e b8.

8.4. Conversão de Dados Numéricos para String

O método estático **valueOf** da classe String permite obter uma String que representa um dado tipo numérico.

Exemplo:

```
double a = 3.1415927;
int i = 123;
System.out.println( "a = " + String.valueOf( a ) );
System.out.println( "i = " + String.valueOf( i ) );
```

Este código resulta na impressão de a = 3.1415927 seguido de i = 123.

O mesmo resultado pode ser obtido com uma sintaxe simplificada, pois numa concatenação de String com o operador +, números são transformados automaticamente em String. Assim, as dua últimas linhas acima podem ser substituidas por:

```
System.out.println( "a = " + a );
System.out.println( "i = " + i );
```

8.5. Conversão de String para Dados Numéricos

Para realizar a operação inversa, ou seja transformar Strings em tipos numéricos, podemos usar as chamadas "wrapper classes", que podem ser encontradas no pacote java.lang.

Por exemplo, a classe wrapper de um tipo int é Integer, double é Double, etc. Para transformar uma String num doublé, pode-se usar o método estático **valueOf** em conjunto com o método **doubleValue** para transformar no tipo primitivo double. Ainda precisaremos "desembrulhar" o nosso número, ou seja traduzir o nosso objeto Double num tipo primitivo double. Isto é feito chamando o método doubleValue da classe

Double. O trecho de código abaixo ilustra este procedimento, assim como o semelhante para o caso de um tipo inteiro. (Outros tipos podem ser tratados também da mesma maneira.)

```
String s1 = "3.1415927";
String s2 = "123";
double d1 = Double.valueOf( s1 ).doubleValue();
double d2 = Double.parseDouble(s1);
Double d3 = new Double(s1);
Double d4 = Double.valueOf( s1 );
int i1 = Integer.valueOf( s2 ).intValue();
int i2 = Integer.parseInt( s2 );
```

Ao executar o código acima as variáveis ficarão assim:

```
d1 = 3.1415927
d2 = 3.1415927
d3 = 3.1415927
d4 = 3.1415927
i1 = 123
i2 = 123
```

8.6. Formatação de String

A transformação de números em String resulta numa representação completa do número, sem arredondamento nem controle do formato. Recursos para formatar texto, e em especial números, podem ser encontrados no pacote java.text. Formatos são objetos em Java. O exemplo a seguir apresenta o uso de um objeto de formatação da classe DecimalFormat:

```
import java.text.*;
…
double a = 3.1415927;
DecimalFormat formato = new DecimalFormat( "0.####" );
System.out.println( "a = " + meuFormato.format( a ) );
```

Saída: a = 3,1416

OBS: o método format faz o arredondamento para o número de casas decimais especificados no formato.

Exemplo para formatar um valor para duas casas decimais, no formato de moeda:

```
DecimalFormat formata = new DecimalFormat("R$ ###,###,##0.00");
double b = 134.5;
System.out.println( "Valor: "+formata.format(b) );
```

Saída: Valor: R$ 134,50

8.7. Decomposição de Strings

Se uma String representa uma sucessão de palavras, pode ser necessário decompô-la em palavras individuais.

A método split() do pacote java.lang permite realizar este tipo de operação. Esse método recebe como parâmetro de entrada uma expressão regular e retorna um array de strings.

Exemplo:

```
String dados = "Real-How-To";
String[] vetor = dados.split("-");
   for (int i = 0 ; i < vetor.length ; i++) {
       System.out.println(vetor[i]);
}
```

Valores das variáveis após a execução:

```
Real
How
To
```

8.8. Mais Exemplos da Classe String

Mais alguns exemplos de uso de métodos da classe String:

```
String texto = "classe ????"; // declara e inicializa a String texto
texto = texto.substring(0,6); // parte da String(posição inicial 0, até a 6)
System.out.println(texto);
System.out.println(texto.toUpperCase()); // converte String p/ MAIÚSCULO
texto += " String"; // concatenar
System.out.println(texto);
int tamanho = texto.length(); // tamanho da String
System.out.println("Tamanho da String: "+tamanho);
```

A saída gerada pelo programa acima é:

```
classe
CLASSE
classe String
Tamanho da String: 13
```

Há outros métodos que permitem trabalhar com Strings, por exemplo, descobrir qual o caráter numa dada posição, entre outras operações úteis. A figura a seguir apresenta diversas funções existentes na classse String:

int	**lastIndexOf**(**String** str) Returns the index within this string of the last occurrence of the specified substring.
int	**lastIndexOf**(**String** str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
int	**length**() Returns the length of this string.
String	**replace**(char oldChar, char newChar) Returns a string resulting from replacing all occurrences of oldChar in this string with newChar.
String	**replaceAll**(**String** regex, **String** replacement) Replaces each substring of this string that matches the given **regular expression** with the given replacement.
String	**replaceFirst**(**String** regex, **String** replacement) Replaces the first substring of this string that matches the given **regular expression** with the given replacement.
String[]	**split**(**String** regex) Splits this string around matches of the given **regular expression**.
String[]	**split**(**String** regex, int limit) Splits this string around matches of the given **regular expression**.
String	**substring**(int beginIndex) Returns a string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a string that is a substring of this string.
char[]	**toCharArray**() Converts this string to a new character array.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**trim**() Returns a string whose value is this string, with any leading and trailing whitespace removed.

Figura 26 - Classe String

9. DATA E HORA - CLASSE DATE

9.1.1. A Classe Date

Em Java informações de data e hora são representadas pela classe Date. A classe Date, encontrada no pacote java.util, encapsula um valor long que representa um momento específico no tempo. Um construtor útil é Date(), que cria um objeto Date representando a hora em que o objeto foi criado. O método getTime() retorna o valor long de um objeto Date.

O exemplo abaixo, ilustra o uso do construtor Date() para criar uma data e inicializar com a data e hora em que o objeto foi criado. O data e hora é obtida do sistema operacional onde o programa está executando. O método getTime() retorna o número de milisegundos que a data representa.

```
import java.util.*;

public class HoraAtual {
   public static void main(String[] args) {
      Date agora = new Date();
      long agoraLong = agora.getTime();
      System.out.println("O valor é " + agoraLong);
   }
}
```

9.1.2. A Classe SimpleDateFormat

A classe SimpleDateFormat possibilita criar Strings que representam formatos em que se deseja trabalhar com data e hora. Alguém nos Estados Unidos pode preferir ver "December 25, 2000", enquanto que na França as pessoas estão mais acostumadas a "25 decembre 2000". Também é muito comum a data set trabalhada no Brasil no formato “25/12/2009 14:35:22”, ou seja, no formato “dd/MM/yyyy hh:mm:ss”.

Para isso, é preciso criar uma instância de uma classe SimpleDateFormat. Este objeto contém informação a respeito de um formato particular no qual a data será tratada. Para usar o formato default do computador, pode-se aplicar o método getDateInstance() para criar o objeto DateFormat apropriado, como apresentado no exemplo abaixo:

```
SimpleDateFormat formatoData = SimpleDateFormat.getInstance();
```

A classe SimpleDateFormat é encontrada no pacote java.text.

9.2. Convertendo Date para String

Pode-se utilizar o método **format** para converter um objeto Date para uma string, conforme exemplo abaixo:

```
        /*
         * Definir uma variável de método para conter
         * uma data de nascimento
         */
        Date diaNascimento;
        // incializar a data com a data do sistema
        diaNascimento = new Date();
```

```
        // vamos criar um objeto de formatação para data
        SimpleDateFormat formatoData = new SimpleDateFormat("dd/MM/yyyy");

        // agora vamos mostrar a data formatada
        System.out.println("A data formatada é: "+ formatoData.format(diaNascimento));
```

O mesmo pode ser feito com a hora, conforme exemplo abaixo:

```
       // vamos criar um objeto para formatar a hora
        SimpleDateFormat formatoHora = new SimpleDateFormat("HH:mm:ss");
        System.out.println("A hora é: "+ formatoHora.format(diaNascimento));
```

9.3. Convertendo String para Date

Pode-se utilizar o método **parse** para converter uma String para um objeto Date, conforme exemplo abaixo:

```
        // vamos criar um objeto de formatação para data
        SimpleDateFormat formatoData = new SimpleDateFormat("dd/MM/yyyy");

        // agora vamos montar uma data a partir de uma string
        String dataLida = "15/02/1987";
        try {
            // vamos agora converter a string para o tipo date
            diaNascimento = formatoData.parse(dataLida);
        } catch (ParseException ex) {
            System.out.println("Não foi possível converter a string para data!");
        }
```

Pode-se utilizar o método **parse** para converter uma String para um objeto Date, para armazenar uma informação de hora, conforme exemplo abaixo:

```
       // vamos criar um objeto para formatar a hora
        SimpleDateFormat formatoHora = new SimpleDateFormat("HH:mm:ss");

        String horaLida = "20:32:40";
        try {
            Date hora = formatoHora.parse(horaLida);
        } catch (ParseException ex) {
            System.out.println("Erro na conversão!");
        }
```

Pode-se utilizar o método **parse** para converter uma String para um objeto Date, para armazenar uma informação de data e hora, conforme exemplo abaixo:

```
       // vamos criar um objeto para formatar a data e hora
        SimpleDateFormat formatoDataHora = new SimpleDateFormat("dd/MM/yyyy  HH:mm:ss");

        String dataHoraLida = "15/02/1987 20:32:40";
        try {
            Date dataHora = formatoDataHora.parse(dataHoraLida);
        } catch (ParseException ex) {
            System.out.println("Erro na conversão!");
        }
```

10. ARRAYS E MATRIZES

Arrays são utilizados para agrupar variáveis do mesmo tipo. O tipo pode ser qualquer tipo primitivo ou qualquer classe de objetos. Não é possível armazenar diferentes tipos em um único array. É possível ter um array de inteiros, ou um array de Strings, ou um array de array, etc.

10.1. Array

10.1.1. Declarar e Criar um Array

Para declarar um array, acrescenta-se colchetes após o nome da variável.

```
int c[];
```

Para criar o array utiliza-se a palavra-chave new e indicando o número de elementos do array entre colchetes:

```
c = new int[ 12 ];
```

Também pode-se declarar e já criar o array na mesma instrução, como a seguir:

```
int c[] = new int[ 12 ];
```

10.1.2. Inicialização de um Array com Tipos Primitivos

Quando um array é criado usando o operador new, todos os índices são inicializados (0 para arrays numéricos, falso para boolean, '\0' para caracteres, e null para objetos).

O índice para especificar um elemento de um array é contado a partir de 0. Por exemplo, os comandos abaixo declaram e inicializam um array de 4 posições:

```
int b[] = new int[ 4 ];
b[ 0 ] = 5;
b[ 1 ] = 8;
b[ 2 ] = 3;
b[ 3 ] = 2;
```

O array pode ser criado e também inicializado através de uma lista de valores entre chaves. Os comandos acima podem ser substituídos por:

```
int b[] = { 5, 8, 3, 2 };
```

10.1.3. Inicialização de um Array com Objetos

No caso de um array de objetos, deve-se considerar que a criação do array não cria os objetos do array. Cria simplesmente uma lista de referências que ainda não apontam para nada (null).

Os objetos ainda precisam ser criados e as suas referências atribuídas aos elementos do array.

Por exemplo, o código a seguir cria um array de 3 posições e adiciona um objeto do tipo Date em cada posição:

```
Date datas[] = new Date[ 3 ];
datas[ 0 ] = new Date();
datas[ 1 ] = new Date("02/25/2007");
datas[ 2 ] = new Date("02/28/2007 21:14:30");
```

10.1.4. Comprimento de um Array

Qualquer array possui uma variável inteira **length** que fornece o número de elementos do array.

No exemplo abaixo, ao percorrer o array, utiliza-se length para determinar o seu tamanho, e assim, a condição de parada:

```
String palavras[] = new String[2];
palavras[0] = "palavra1";
palavras[1] = "palavra2";
// percorrer o array
for(int pos=0; pos < palavras.length; pos++)
     System.out.println("Índice "+pos+" : "+palavras[pos]);
```

Saída:

```
Índice 0 : palavra1
Índice 1 : palavra2
```

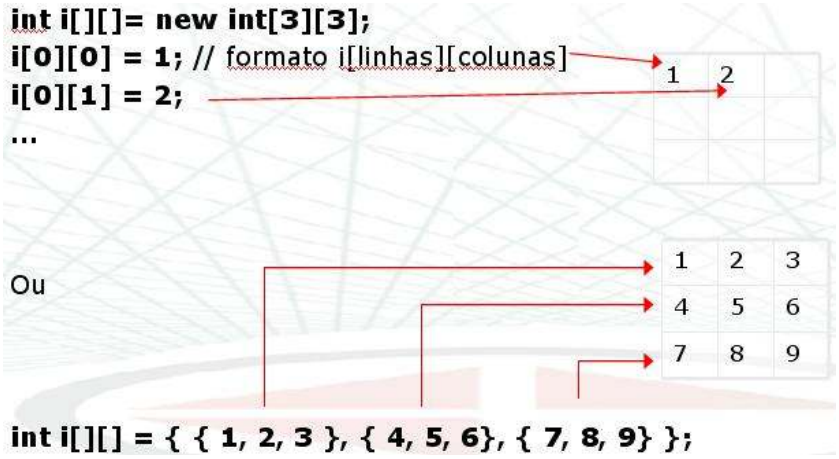
10.1.5. Mais Exemplos de Array

```
public class ExemploArray {
    public static void main(String[] args) {
       // Array de char
       char[] alfabeto = new char[26];
       alfabeto[0] = 'A';
       alfabeto[1] = 'B';   // ...
       alfabeto[25] = 'Z';
       // ou
       char[] alfabeto2 = {'A', 'B', '.', '.', '.', 'Z'};
       // Array de String
       String palavras[] = new String[2];
       palavras[0] = "palavra1";
       palavras[1] = "palavra2";
       // ou
       String[] palavras2 = new String[] { "outra", "forma", "de",
"inicializar" };
       // Array de int
       int[] a = new int[3];
       a[0] = 1;
       a[1] = 2;
       a[2] = 3;
       // ou
       int[] b = { 1, 2, 3 };
       // percorrer o array
       for(int pos=0; pos<palavras.length; pos++)
            System.out.println(palavras[pos]);
    }
}
```

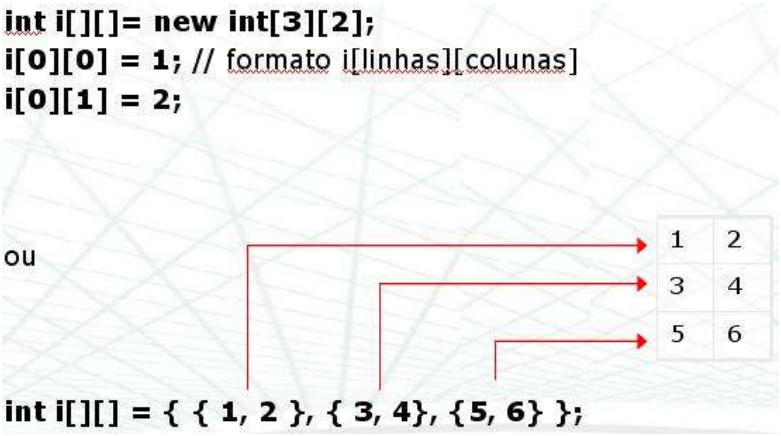
10.2. Matrizes

Arrays de múltiplos índices podem ser definidos, sendo considerados como **arrays de arrays**. Os seguintes exemplos criam objetos que são essencialmente matrizes:

Exemplo 1:



Exemplo 2:



10.2.1. Mais Exemplos de Matrizes

```
public class ExemplosMatriz {
    public static void main(String[] args) {
        int[][] matriz = new int[2][2];
        matriz[0][0] = 1;
        matriz[0][1] = 2;
        matriz[1][0] = 3;
        matriz[1][1] = 4;
        for (int i=0; i < matriz.length; i++)
        {
            for (int j=0; j < matriz[i].length; j++)
                System.out.println("["+i+","+j+"]="+matriz[i][j]);
        }
        System.out.println();
        int[][]  matriz2 = { {5,6},{7,8} };
        for (int i=0; i < matriz2.length; i++)
        {
            for (int j=0; j < matriz2[i].length; j++)
                System.out.println("["+i+","+j+"]="+matriz2[i][j]);
        }
    }
}
```

Saída:

```
[0,0]=1
[0,1]=2
[1,0]=3
[1,1]=4

[0,0]=5
[0,1]=6
[1,0]=7
[1,1]=8
```

10.3. Passagem por Referência

Um array pode ser passado como argumento a um método. Deve-se notar que array são objetos e, na linguagem Java, objetos são sempre passados aos métodos por referência, ao passo que tipos primitivos são sempre passados por valor. Isto quer dizer que uma modificação ao array realizada no método chamado implicará na mesma modificação ao array definido no programa que chamou o método em questão. Veja no exemplo a seguir:

```
public static void main(String[] args) {
    // TODO code application logic here
    int v[] = new int[2];
    v[0] = 10;
    v[1] = 5;
    modifica(v);
    System.out.println( v[1] );
}

public static void modifica(int vet[]){
    vet[1] = 44;
}
```

A saída será 44

10.4. Exercício sobre Array

Ler dois arrays de 10 números (A e B) e:

- calcular S = (A[0] * B[9]) + (A[1] * B[8]) + ...
- calcular C, sendo C[i] = A[i] / B[i]
- imprimir os números pares de A
- imprimir o valor de S
- imprimir os números de C

- Controlar para que os números lidos sejam inteiros. Se não informar inteiro, ler o número novamente.

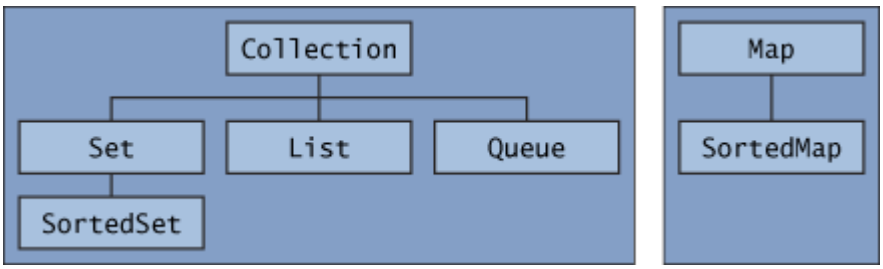
- Controlar para que se houver algum erro durante o cálculo, seja mostrada uma mensagem adequada ao usuário.

- Pode usar System.out.println() para gerar a saída.

11. COLEÇÕES

As coleções podem armazenar um número arbitrário de elementos, sendo que cada elemento é um outro objeto. O Java contém um framework (Java Collection Framework) com uma API de coleções do Java, organizada em uma hierarquia de interfaces e em uma hierarquia de implementação de classes em separado. **Interfaces:** permitem que as coleções sejam manipuladas independentes de suas implementações; **Implementações:** Classes que implementam uma ou mais interfaces.

O diagrama abaixo demonstra as interfaces de coleções no Java são organizadas:



Collection: Não existe uma implementação direta desta *inferface*, porém, ela está no topo da hierarquia definindo operações que são comuns a todas as coleções;

Set: Está diretamente relacionada com a idéia de conjuntos. Assim como um conjunto, as classes que implementam esta interface não podem conter elementos repetidos. Podem ser usadas implementações de *SortedSet* para situações onde for necessário ordenar os elementos;

List: Também chamada de seqüência. É uma coleção ordenada, que ao contrário da inferface *Set*, pode conter valores duplicados. Além disso, permite o controle total sobre a posição onde se encontra cada elemento da coleção, podendo acessar cada um deles pelo índice.

Queue: Normalmente utilizada quando for necessárioa uma coleção do tipo FIFO (First-In-First-Out), também conhecida como fila.

Map: Utilizada quando for necessária uma relação de chave-valor entre os elementos. Cada chave pode conter apenas um único valor associado. Utiliza-se *SortedMap* para situações onde for necessário ordenar os elementos.

A seguir são apresentadas classes de implementação de cada interface:

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

11.1. HashSet

HashSet é uma classe de implementação para coleções da interface Set.

```
import java.util.*;
public class ExHashSet {
  public static void main(String[] args) {
       HashSet lista = new HashSet ();  // criar um HashSet
       lista.add("Linha 1"); // Adiciona uma String na coleção
       lista.add(new Date()); // Adiciona uma data na coleção

       for(Object o : lista) //percorre a coleção recuperando os objetos
         System.out.println(o);
   }
}
```

11.2. ArrayList

ArrayList é uma classe de implementação para coleções da interface List.

```
import java.util.*;
public class ExArrayList {
  public static void main(String[] args) {
       ArrayList lista = new ArrayList();  // criar um ArrayList
       lista.add("Linha 1"); // Adiciona uma String na coleção
       lista.add(new Date()); // Adiciona uma data na coleção

       for(Object o : lista) //percorre a coleção recuperando os objetos
         System.out.println(o);
   }
}
```

11.3. Vector

Semelhante ao ArrayList, uma classe Vector também realiza uma implementação da interface List, podendo ser usada de forma semelhante ao ArrayList. O exemplo a seguir ilustra como utilizar uma classe Vector.

```
import java.util.*;
public class ExVector {
  public static void main(String[] args) {
       Vector lista = new Vector();  // criar um Vector
       lista.add("Linha 1"); // Adiciona uma String na coleção
       lista.add(new Date()); // Adiciona uma data na coleção

       for(Object o : lista) //percorre a coleção recuperando os objetos
         System.out.println(o);
   }
}
```

11.4. Iterator

Um Iterador é um objeto que fornece funcionalidade para iterar por todos os elementos de uma coleção. O exemplo a seguir ilustra como adicionar e remover objetos da classe ArrayList e o uso do Iterator para percorrer todo o ArrayList e imprimir todos os objetos.

```
import java.util.*;
public class ExArrayList {
  public static void main(String[] args) {
    ArrayList objetos = new ArrayList();  // criar um ArrayList

    objetos.add("Primeiro"); // Adiciona um objeto String
    objetos.add(new Date()); // Adiciona um objeto Date
    objetos.add(new Double(10.45)); // Adiciona um objeto Double

    Iterator it = objetos.iterator(); // objeto para percorrer o ArrayList
    while(it.hasNext()) {
        System.out.println("Lista -> "+it.next()); // imprime o objeto
    }
  }
}
```

O exemplo a seguir ilustra como adicionar e remover objetos da classe ArrayList. Os objetos utilizados são gerados a partir de uma classe Cliente.

```
import java.util.*;
public class ExArrayList2 {
    public static void main(String[] args) {
        ArrayList objetos = new ArrayList();  // cria o ArrayList
        Cliente c1 = new Cliente(); // cria o objeto c1 da classe Cliente
        Cliente c2 = new Cliente(); // cria o objeto c2 da classe Cliente
        c1.nome = "Cliente 1"; // inicializa atributo nome de c1
        c2.nome = "Cliente 2"; // inicializa atributo nome de c2

        objetos.add(c1); // adicionar c1 no ArrayList
        objetos.add(c2); // adicionar c2 no ArrayList
        Cliente cr1 = (Cliente) objetos.get(1); // recupera o objeto da pos 1
        System.out.println("O cliente é "+ cr1.nome);

        Iterator it = objetos.iterator(); // para pegar todos os objetos
        Cliente cr2;
        while(it.hasNext()){
            cr2 = (Cliente) it.next(); // recupera o objeto da posição
            System.out.println("Lista -> "+cr2.nome);
        }
    }
}
class Cliente { // definição da classe Cliente
    String nome; // atributo da classe cliente
}
```

11.5. Métodos para Manipular Coleções

As classes de coleções possuem métodos que premitem realizam uma série de operações sobre a coleção, tais como:

add(Object obj): adicionar um objeto na coleção
addAll(Collection<Object> obj): adicionar uma sub-coleção de objetos na coleção
clear(): limpar o conteúdo de uma coleção;
remove(int index): remover um objeto pelo índice de sua posição;
remove(Object obj): remover um objeto pela sua referência;

11.6. Coleções Tipadas – Uso de Generics

Antes dos tipos genéricos (isto é antes do Java 5.0), o compilador não se importava com o que era inserido em um conjunto como o ArrayList por exemplo, porque todas as implementações de conjuntos eram declaradas para conter o tipo Object.

Como vimos nos exemplos acima, poderíamos inserir qualquer coisa em qualquer ArrayList, isso era possível por que ao armazenar o objeto no ArrayList ele o guardava em uma variável de referencia do tipo Object, a qual como sabemos é a superclasse de que se derivam todas as demais classes do java, e justamente por causa deste "excesso de compatibilidade" dos objetos para com o ArrayList era necessária a conversão (cast) dos objetos retornados do ArrayList para que pudéssemos reutiliza-los.

Embora os tipos genéricos possam ser usados de outras maneiras, sua principal finalidade é permitir a criação de conjuntos com compatibilidade de tipos. Ou seja, dessa maneira o compilador o impede de inserir um objeto Dog em uma lista de objetos Duck por exemplo, e graças a declaração do tipo de objetos que nosso conjunto armazenará e já retornará o objeto correto sem a necessidade cast.

E qual a vantagem disso? Agora com os tipos genéricos, podemos inserir somente objetos Duck em um ArrayList<Duck> para que eles saiam com esse mesmo tipo de referencia, deixando assim o programador livre de se preocupar com o fato de alguém inserir um objeto Dog na lista de patos, ou ainda de capturarmos algo que não possa ser convertido em uma referencia de Duck, aumentando ainda mais a segurança da linguagem, pois todos os possíveis erros citados somente seriam descobertos em tempo de execução, e agora com os tipos genéricos todos estes problemas já são detectados na compilação do código.

Porem se você gosta de viver perigosamente ainda poderá declarar um ArrayList que possa armazenar qualquer tipo de objetos, o declarando como ArrayList<Object> que funcionará da mesma maneira que o antigo ArrayList.

No exemplo anterior sobre ArrayList, onde armazenamos um objeto Cliente em uma ArrayList antiga tínhamos que modelar para um formato de cliente todos os objetos que resgatávamos de nossa lista. Vejamos como ficaria nosso exemplo utilizando as novas listas implementadas a partir do java 5.

```
import java.util.*;
public class ExArrayList3 {
    public static void main(String[] args) {
        ArrayList<Cliente> objetos = new ArrayList<Cliente>();   // cria o ArrayList do tipo
Cliente
        Cliente c1 = new Cliente(); // cria o objeto c1 da classe Cliente
        Cliente c2 = new Cliente(); // cria o objeto c2 da classe Cliente
        c1.nome = "Cliente 1"; // inicializa atributo nome de c1
        c2.nome = "Cliente 2"; // inicializa atributo nome de c2

        objetos.add(c1); // adicionar c1 no ArrayList<Cliente>
        objetos.add(c2); // adicionar c2 no ArrayList<Cliente>
        Cliente cr1 = objetos.get(1); // recupera o cliente, e o atribui sem a necessidade de
cast
        System.out.println("O cliente é "+ cr1.nome);
    }
}
class Cliente { // definição da classe Cliente
    String nome; // atributo da classe cliente
}
```

Mas lembre-se a classe ArrayList não foi a única que sofreu mudanças, outras classes de coleções como HashMap, TreeSet, LinkedHashMa, etc., também aderiram ao novo modelo, por isso para saber quais classes você pode utilizar com este novo formato aconselhamos consultar a documentação.

11.7. Exercício ArrayList

Altere o programa ExArrayList2 acrescentando na classe Cliente os atributos telefone e cidade, inicializando os atributos para ambos os clientes. Mostrar na listagem todos os atributos da classe Cliente.

11.8. Exercício usando coleção

Ler o Código, o Nome, Data de Nascimento e o Salário de 5 Pessoas. Ler um valor Inteiro que será o percentual de reajuste do salário que foi informado para cada pessoa.

Recalcular o salário e mostrar a relação atualizada com: Código, Nome, Data de Nascimento e Novo Salário.

12. A CLASSE CALENDAR

A classe abstrata Calendar encapsula um momento no tempo representado em milissegundos. Também provê métodos para manipulação desse momento. A subclasse concreta de Calendar mais usada é a GregorianCalendar que representa o calendário usado pela maior parte dos países.

Para obter um Calendar que encapsula o instante atual (data e hora), usamos o método estático **getInstance**() de **Calendar**. A partir de um Calendar, podemos saber o valor de seus campos, como ano, mês, dia, hora e minuto. Para isso, usamos o método **get()** que recebe um inteiro representando o campo.

Os valores possíveis estão em constantes na classe Calendar. Para trabalhar com dia, mês e ano, pode-se utilizar as constantes DAY_OF_MONTH, MONTH e YEAR respectivamente. Veja o exemplo a seguir:

```
import java.util.Calendar;
public class TestesCompleto {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data atual do SO
        Calendar c = Calendar.getInstance();
        // imprimir dia/mes/ano
        System.out.println(c.get(Calendar.DAY_OF_MONTH)+"/"+
                           (c.get(Calendar.MONTH)+1)+"/"+
                            c.get(Calendar.YEAR));
    }
}
```

No exemplo acima, primeiramente é criada uma instância de Calendar chamada c. Em seguida, utiliza-se o método get() para obter o dia, mês e ano para imprimir a data. Note que o mês inicia em 0 para Janeiro até 11 para Dezembro.

Também é possível especificar uma data usando o método set. Veja o exemplo a seguir:

```
import java.util.Calendar;
public class EspecificarData {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data do SO
        Calendar c = Calendar.getInstance();// pegar data atual
        // setar dia/mes/ano para 11 de abril de 2007
        c.set(Calendar.DAY_OF_MONTH,11);
        c.set(Calendar.MONTH,3); // 0-Janeiro
        c.set(Calendar.YEAR,2007);
        System.out.println(c.get(Calendar.DAY_OF_MONTH)+"/"+
                           (c.get(Calendar.MONTH)+1)+"/"+
                            c.get(Calendar.YEAR)); // imprime dia/mês/ano
    }
}
```

É possível incrementar ou decrementar o dia, mês ou ano, usando o método add(), passando um inteiro positivo para incremento e negativo para decremento. Veja o exemplo a seguir:

```
import java.util.Calendar;
public class IncrementarDecrementar {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data do SO
        Calendar c = Calendar.getInstance();// pegar data atual
        // incrementar ou decrementar dia/mes/ano
        c.add(Calendar.DATE,-1); // decrementa um dia
        c.add(Calendar.MONTH,1); // incrementa um mês
        c.add(Calendar.YEAR,2); // imcrementa dois anos
        System.out.println(c.get(Calendar.DAY_OF_MONTH)+"/"+
                           (c.get(Calendar.MONTH)+1)+"/"+
                            c.get(Calendar.YEAR)); // imprime dia/mês/ano
    }
}
```

Para trabalhar com hora, minuto e segundos, pode-se utilizar as constantes HOUR_OF_DAY, MINUTE e SECOND respectivamente. A constrante HOUR_OF_DAY trabalhar a hora de 0 a 24. Também pode-se utilizar para hora as constantes HOUR e AM_PM para trabalhar no formado 0-12 AM/PM.

Veja o exemplo a seguir:

```
import java.util.Calendar;
public class ImprimiHoraMinSeg {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data do SO
        Calendar c = Calendar.getInstance();// pegar data atual
        // imprimir Hora:Min:Seg
        System.out.println(c.get(Calendar.HOUR_OF_DAY)+":"+
                           c.get(Calendar.MINUTE)+":"+
                           c.get(Calendar.SECOND));
        // ou
        String am_pm[] = {"AM", "PM"};
        System.out.println(c.get(Calendar.HOUR)+":"+
                           c.get(Calendar.MINUTE)+":"+
                           c.get(Calendar.SECOND)+" "+
                           am_pm[c.get(Calendar.AM_PM)]);
    }
}
```

Também é possível setar, incrementar e decrementar hora, minuto e segundo como o uso do método add(). Veja o exemplo a seguir:

```
import java.util.Calendar;
public class SetarImcrementoDecremento {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data do SO
        Calendar c = Calendar.getInstance();// pegar data atual
        // setar hora, min e seg
        c.set(Calendar.HOUR,8); // 00-12
        c.set(Calendar.AM_PM,Calendar.PM); // PM
        // imprimir Hora:Min:Seg
        String am_pm[] = {"AM", "PM"};
        System.out.println(c.get(Calendar.HOUR)+":"+
                           c.get(Calendar.MINUTE)+":"+
                           c.get(Calendar.SECOND)+" "+
                           am_pm[c.get(Calendar.AM_PM)]);
        // setar hora, min e seg
        c.set(Calendar.HOUR_OF_DAY,17); // 00-23
        c.set(Calendar.MINUTE,30); // 00-59
        // imprimir Hora:Min:Seg
        System.out.println(c.get(Calendar.HOUR_OF_DAY)+":"+
                           c.get(Calendar.MINUTE)+":"+
                           c.get(Calendar.SECOND));
    }
        }
```