Fundamentos

Agenda

- Tipos Primitivos
- Palavras-Chave e Palavras Reservadas
- Classes e Objetos
- Conversão entre Tipos
- Constantes
- Operadores
- Funções ou sub-rotinas
- Controle de Fluxo
- Vetores e Coleções
- Exceções

Tipos Primitivos

- Fortemente tipada
- Oito (8) tipos primitivos
  - Seis (6) tipos primitivos numéricos
    - Quatro (4) tipos primitivos inteiros
    - Dois (2) tipos primitivos de ponto flutuante
  - Um (1) tipo primitivo caracter
  - Um (1) tipo primitivo booleano

Tipos Primitivos Numéricos

Tipos primitivos Inteiros

Tipo	Tamanho	Val. mínimo	Val. máximo
byte	1 byte	-128	127
short	2 bytes	-32.768	32.767
int	4 bytes	-2.147.483.648	2.147.483.647
long	8 bytes	-9.223.372.036.854.775.808	9.223.372.036.854.775.807

Literais Inteiros

- Podem ser expressos em formato decimal, octal ou hexadecimal
- O formato padrão é decimal
  Ex.: **`28`**
- Para octal, o literal deve ser precedido por 0 (zero)
  Ex.: **`034`**
- Para hexadecimal, o literal deve ser precedido por 0x ou 0X

  Ex.: **`0x1c , 0x1C , 0X1c , 0X1C`**
- Para binário, o literal deve ser precedido de 0b

  Ex.: **`0b0111_0001 , 0b010101`**

Tipos Primitivos Numéricos

Tipos Primitivos de Ponto Flutuante

Tipo	Tamanho	Intervalo
float	4 bytes	Aproximadamente +/- 3.40282347E+38F (6-7 dígitos significativos)
double	8 bytes	Aproximadamente +/- 1.79769313486231570E+308 (15 dígitos significativos)

Literais Ponto-Flutuante

- Para que um literal numérico seja interpretado como um valor em ponto-flutuante, este deve encaixar-se em um dos seguintes casos:
  - Conter um ponto decimal

    Ex.: **1.414**
  - Conter a letra **E**, indicando notação científica

    Ex.: **4.23E+21**
  - Conter o sufixo **F** ou **f**, indicando um literal float

    Ex.: **1.828f**
  - Conter o sufixo **D** ou **d**, indicando um literal double

    Ex.: **1234d**
- Um literal ponto-flutuante sem prefixo **F** ou **D** é interpretado como um literal double

Tipo Primitivo Caractere

char

- Utiliza-se apóstrofe para se representar constantes char
- Representa caracteres segundo o esquema Unicode (código de 2 bytes que permite a representação de 65.536 caracteres diferentes)

Ex:

```
char c = 'w';
char c1 = '\u4567';
```

Tipo Primitivo Booleano

boolean

- É empregado em testes lógicos usando operadores relacionais que a linguagem Java suporta

- Valores possíveis:
  - true
  - false

Ex.:

```
boolean clienteEspecial = true;
boolean emDebito = false;
```

Caracteres Especiais

Caracteres especiais

Seq. contr.	Nome	Valor Unicode
\b	backspace	\u0008
\t	tab	\u0009
\n	linefeed	\u000a
\r	carriage return	\u000d
\”	aspas	\u0022
\'	apóstrofe	\u0027
\\	barra invertida	\u005c

Variáveis

Palavras utilizadas para nomear variáveis, métodos e classes

- Devem ser iniciados por letra, dólar ($), seguidos por zero ou mais letras, dólar ou dígitos
- Não podem ser palavras-chave ou palavras-reservadas
- Identificadores são sensíveis a letras maiúsculas ou minúsculas

Variáveis

- Ex. nomes válidos:

```
byte b21;
int umaVarialvelInteira;
long $umaVariavelLong;
char ch;
```

- Ex. nomes inválidos:

```
byte !bi;
char @letra;
double 43_variavel;
```

Palavras-Chave e Palavras Reservadas

abstract
assert
boolean
break
byte
case
catch
char
class
const
continue
default
do
double
else
long
native
new
null
package
private
protected
public
return
short
static
strictfp
super
switch
synchronized
this
throw
throws
transient
true
try
var
void
volatile
while

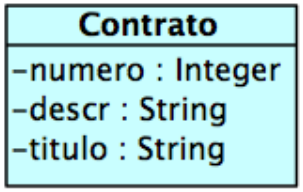
Atribuições e Inicializações

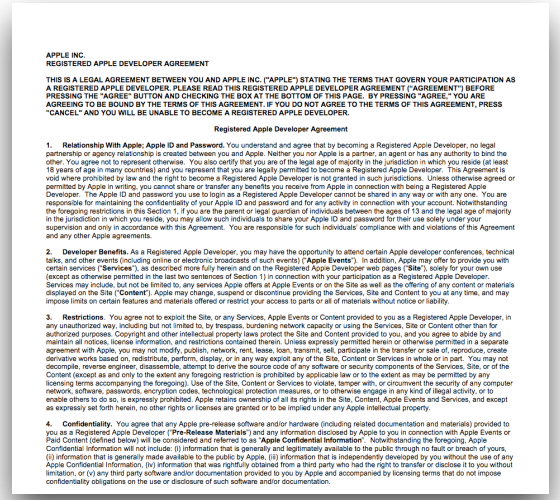
```
int foo;
foo = 37;

int i = 10;
long l = 10, j = 20;

char charSim;
charSim = 'S';
```

Classes e Objetos

Classes são estruturas de dados que representam abstrações de Objetos do mundo real.

Estas abstrações são representadas na forma de atributos, necessários ao armazenamento da informações que serão utilizadas nas aplicações.





APPLE INC.
REGISTERED APPLE DEVELOPER AGREEMENT

THIS IS A LEGAL AGREEMENT BETWEEN YOU AND APPLE INC. ("APPLE") STATING THE TERMS THAT GOVERN YOUR PARTICIPATION AS A REGISTERED APPLE DEVELOPER. PLEASE READ THIS REGISTERED APPLE DEVELOPER AGREEMENT ("AGREEMENT") BEFORE PRESSING THE "AGREE" BUTTON AND CHECKING THE BOX AT THE BOTTOM OF THIS PAGE. BY PRESSING "AGREE," YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, PRESS "CANCEL" AND YOU WILL BE UNABLE TO BECOME A REGISTERED APPLE DEVELOPER.

Registered Apple Developer Agreement

1. **Relationship With Apple; Apple ID and Password.** You understand and agree that by becoming a Registered Apple Developer, no legal partnership or agency relationship is created between you and Apple. Neither you nor Apple is a partner, an agent or has any authority to bind the other. You agree not to represent otherwise. You also certify that you are of the legal age of majority in the jurisdiction in which you reside (at least 18 years of age in many countries) and you represent that you are legally permitted to become a Registered Apple Developer. This Agreement is void where prohibited by law and the right to become a Registered Apple Developer is not granted in such jurisdictions. Unless otherwise agreed or permitted by Apple in writing, you cannot share or transfer any benefits you receive from Apple in connection with being a Registered Apple Developer. The Apple ID and password you use to login as a Registered Apple Developer cannot be shared in any way or with any one. You are responsible for maintaining the confidentiality of your Apple ID and password and for any activity in connection with your account. Notwithstanding the foregoing restrictions in this Section 1, if you are the parent or legal guardian of individuals between the ages of 13 and the legal age of majority in the jurisdiction in which you reside, you may allow such individuals to share your Apple ID and password for their use solely under your supervision and only in accordance with this Agreement. You are responsible for such individuals' compliance with and violations of this Agreement and any other Apple agreements.

2. **Developer Benefits.** As a Registered Apple Developer, you may have the opportunity to attend certain Apple developer conferences, technical talks, and other events (including online or electronic broadcasts of such events) ("**Apple Events**"). In addition, Apple may offer to provide you with certain services ("**Services**"), as described more fully herein and on the Registered Apple Developer web pages ("**Site**"), solely for your own use (except as otherwise permitted in the last two sentences of Section 1) in connection with your participation as a Registered Apple Developer. Services may include, but not be limited to, any services Apple offers at Apple Events or on the Site as well as the offering of any content or materials displayed on the Site ("**Content**"). Apple may change, suspend or discontinue providing the Services, Site and Content to you at any time, and may impose limits on certain features and materials offered or restrict your access to parts or all of materials without notice or liability.

3. **Restrictions.** You agree not to exploit the Site, or any Services, Apple Events or Content provided to you as a Registered Apple Developer, in any unauthorized way, including but not limited to, by trespass, burdening network capacity or using the Services, Site or Content other than for authorized purposes. Copyright and other intellectual property laws protect the Site and Content provided to you, and you agree to abide by and maintain all notices, license information, and restrictions contained therein. Unless expressly permitted herein or otherwise permitted in a separate agreement with Apple, you may not modify, publish, network, rent, lease, loan, transmit, sell, participate in the transfer or sale of, reproduce, create derivative works based on, redistribute, perform, display, or in any way exploit any of the Site, Content or Services in whole or in part. You may not decompile, reverse engineer, disassemble, attempt to derive the source code of any software or security components of the Services, Site, or of the Content (except as and only to the extent any foregoing restriction is prohibited by applicable law or to the extent as may be permitted by any licensing terms accompanying the foregoing). Use of the Site, Content or Services to violate, tamper with, or circumvent the security of any computer network, software, passwords, encryption codes, technological protection measures, or to otherwise engage in any kind of illegal activity, or to enable others to do so, is expressly prohibited. Apple retains ownership of all its rights in the Site, Content, Apple Events and Services, and except as expressly set forth herein, no other rights or licenses are granted or to be implied under any Apple intellectual property.

4. **Confidentiality.** You agree that any Apple pre-release software and/or hardware (including related documentation and materials) provided to you as a Registered Apple Developer ("**Pre-Release Materials**") and any information disclosed by Apple to you in connection with Apple Events or Paid Content (defined below) will be considered and referred to as "**Apple Confidential Information**". Notwithstanding the foregoing, Apple Confidential Information will not include: (i) information that is generally and legitimately available to the public through no fault or breach of yours, (ii) information that is generally made available to the public by Apple, (iii) information that is independently developed by you without the use of any Apple Confidential Information, (iv) information that was rightfully obtained from a third party who had the right to transfer or disclose it to you without limitation, or (v) any third party software and/or documentation provided to you by Apple and accompanied by licensing terms that do not impose confidentiality obligations on the use or disclosure of such software and/or documentation.

Classes e Objetos

Uma Classe é declarada contento um **nome** e seus **atributos**.

O nome da classe define um novo **Tipo de Dado** seus atributos representam a estrutura deste tipo.

Os **Objetos** criados a partir desta **classe** herdam a estrutura declarada na classe, desta forma é possível agrupar valores de tipos diferentes sob um único nome.

```
public class Cliente {
    String nome;
    String email;
    int idade;
}
```

```
Cliente novo = new Cliente();

novo.nome = "João da Silva";
novo.email = "jsilva@gmail.com";
novo.idade = 32;
```

Conversões

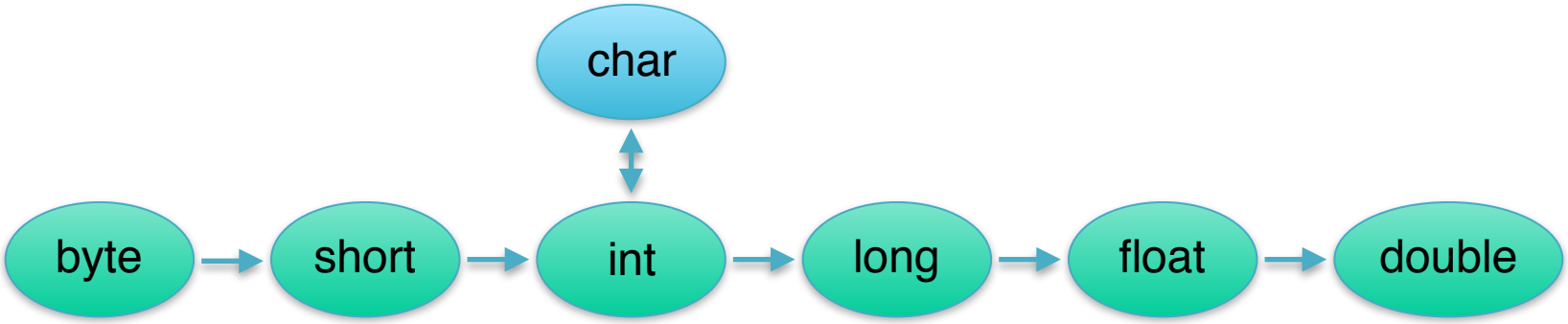
As conversões de tipos podem ser divididas em 2 categorias:

- Conversões Implícitas (Conversão)
- Conversões Explícitas (Cast ou Coerção)

Conversões

As regras gerais para conversão em atribuição de primitivas são:

- Um boolean não pode ser convertido para qualquer outro tipo.
- Um tipo não booleano pode ser convertido para outro tipo não booleano conforme o gráfico a seguir:



Conversões

Tipo a ser convertido	Converter em	Forma de conversão
int x = 10;	float	float y = (float)x;
int x = 10;	double	double y = (double)x;
float x = 10.5f;	int	int y = (int)x
String x = “10”;	int	int y = Integer.parseInt(x);
String x = “20.54”;	float	float y = Float.parseFloat(x);
String x = “20.54”;	double	double y = Double.parseDouble(x);
String x = “Java”;	Array de bytes	byte[] y = x.getBytes();
int x = 10;	String	String y = String.valueOf(x);
float x = 10.5f;	String	String y = String.valueOf(x);
double x = 10.45;	String	String y = String.valueOf(x);
byte[] x = {'J', 'a', 'v', 'a'};	String	String y = new String(x);

A Classe String

```
int it = 5;
String st = String.valueOf(it);

String sujeito = "Sócrates";
String adjetivo = new String("filósofo");
String nome = sujeito + " foi um grande ";
nome += adjetivo;

char[ ] texto = { 'J', 'a', 'v', 'a' };
String titulo = new String(texto);
```

Wrappers

Tipo Primitivo	Classe Wrapper
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

Wrappers

- Cada uma delas serve para representar dados de um dos tipos primitivos na forma de objetos
- As Classes **Boolean** e **Character** derivam diretamente da classe **Object**
- As Classes **Byte**, **Short**, **Integer**, **Long**, **Float** e **Double** derivam de uma classe abstrata chamada **Number**
- Estas classes implementam os métodos **byteValue**(), **shortValue**(), **intValue**(), **longValue**(), **floatValue**() e **doubleValue**() que são utilizados para converter o valor numérico representado em um valor dos tipos **byte**, **short**, **int**, **long**, **float** e **double** respectivamente

Wrappers

- A classe **Boolean** também possui um método para a conversão do valor representado em um tipo primitivo **boolean**, chamado **booleanValue**()
- A classe **Character**, por sua vez, converte seu conteúdo em um tipo **char** através do método **charValue**()
- Um valor numérico que está representado como texto pode ser convertido em um tipo primitivo
- Isso é feito utilizando-se os métodos estáticos **parseByte**(), **parseShort**(), **parseInt**(), **parseLong**(), **parseFloat**() e **parseDouble**() das classes **Byte**, **Short**, **Integer**, **Long**, **Float** e **Double** respectivamente

Wrappers

- A classe **Character** contém os métodos estáticos **getNumericValue**(), **isDigit**(), **isLetter**(), **isLetterOrDigit**(), **isWhiteSpace**(), **isLowerCase**(), **isUpperCase**() e **toUpperCase**()
- Os métodos **is**...() retornam um valor booleano resultante de um teste realizado com um caractere
- Os métodos **to**...() retornam um char convertido para maiúsculo (**Upper**) ou minúsculo (**Lower**)
- O método **getNumericValue**() retorna o valor **int** que representa o código Unicode para o argumento informado

Wrappers

- A classe **BigInteger** pode ser usada para representar números inteiros e contém métodos que substituem o uso de todos os operadores aplicados em operações com inteiros e todos os métodos relevantes da classe **java.Math**

- A classe **BigDecimal**, por sua vez, pode ser usada para representar números de ponto flutuante

- Ela suporta operações aritméticas básicas, manipulação de escala, conversão de formato, bem como a adoção de oito modos de arredondamento diferentes

Constantes

- Em Java, usa-se a palavra-chave `final` para denotar uma constante
- `final` indica que pode ser atribuído valor a uma variável somente uma vez
- É costume dar nomes totalmente em maiúscula à constantes
- Ex.: `final double CM_POR_SOL = 2.54;`

Operadores de Atribuição

- A expressão a direita é atribuída à variável a esquerda

```
int var1 = 0, var2 = 0;
var1 = 50;          // var1 tem valor = 50
var2 = var1 + 10; // var2 tem valor = 60
```

- A expressão a direita é sempre executada antes da atribuição

- Atribuições podem ser agrupadas

```
int var3;
var1 = var2 = var3 = 50;
```

Operadores Aritméticos

- Executa operações aritméticas básicas
- Trabalham com variáveis numéricas e literais

```
int a, b, c, d, e;
a = 2 + 2; // adição
b = a * 3; // multiplicação
c = b - 2; // subtração
d = c / 2; // divisão
e = d % 2; // resto da divisão
```

Incremento e Decremento

- O operador ++ incrementa em 1
- O operador -- decrementa em 1

```
int var1 = 3;
var1++;                    // var1 agora é = 4
```

Incremento e Decremento

- O operador ++ pode ser usado de duas formas

```
int var2 = 3, var3 = 0, var4 = 0;
// pré-fixado:
//    primeiro incrementa var2
//    depois atribui o valor em var3
var3 = ++var2;
// pós-fixado:
//    primeiro atribui o valor em var4
//    depois decrementa var2
var4 = var2--;
```

Atribuição Combinada

- Um operador de atribuição pode ser combinado com qualquer operador aritmético

```
double total = 0, num = 1;
double taxa = 0.5;

total = total + num;  // total é = 1
total += num;         // total é = 2
total -= 5;           // total é = -3
total *= taxa;        // total é = -1.5
```

Precedência

- Existe uma precedência para as operações aritméticas. Abaixo temos a tabela de precedência:

Hierarquia	***Operação***
1	Parênteses
2	Função
3	-, + (unários)
4	%, *, /
5	+, -

Expressões aritméticas

- Segue alguns exemplos de construção de expressões aritméticas:

3 / 4 + 5	= 5.75
3 / (4 + 5)	= 0.3333333
3 / 2 * 9	= 13.5
11 % 3 * 2	= 4
11 % (3 * 2)	= 5
(11 % 3) * 2	= 4
3 / 2 + (65 - 40) * (1 / 2)	= 14

Funções ou Sub-Algoritmos

- São trecho de algoritmos que efetuam um ou mais tarefas
- Podem funcionar em conjunto com qualquer outro algoritmo para resolver um determinado problema
- Pode ser reutilizado em outros pontos do algoritmo
- Reduzem o tamanho do código
- Facilitam a compreensão e visualização do algoritmo
- A “função” sempre deve informar o tipo do valor que retorna

Funções

A “função” é declarada em qualquer parte do programa e sempre retorna apenas um valor ao algoritmo que o chamou

```
public void inicio() {
    String produto = leTexto("Informe o Nome do Produto");
    escrevaL("O produto ", produto,
            " terá um custo de R$", calcula());
}

double calcula() {
    double taxa = 1.15;
    double valor = leReal("Informe o Preço");
    return valor * taxa;
}
```

Funções

Uma “função” pode opcionalmente receber uma lista de argumentos, estes argumentos são visíveis somente para a função

```
public void inicio() {
    String produto = leTexto("Informe o Nome do Produto");
    double preco = leReal("Informe o Preço");
    escrevaL("O produto ", produto,
            " terá um custo de R$", calcula(preco));
}

double calcula(double valor) {
    double taxa = 1.15;
    return valor * taxa;
}
```

Funções

Quando a “função” não retornar valor, o tipo a ser informado deve ser “**void**”

```
public void inicio() {
    String produto = leTexto("Informe o Nome do Produto");
    double preco = leReal("Informe o Preço");
    apresentaCalculo(produto, preco);
}

void apresentaCalculo(String nome, double valor) {
    double taxa = 1.15;
    double total = valor * taxa;
    escrevaL("O produto ", nome,
            " terá um custo de R$", total);
}
```

Função Recursiva

Uma “função” pode requisitar a execução de outra “função” e quando esta execução é para ela mesma, a chamamos de “**função recursiva**”

```
public void inicio() {
    int num = leInteiro("Informe um nº para o calculo de Fibonacci");
    escrevaL("O valor calculado com a função fibonnaci para ",
            num, " é: ", fibonacci(num));
}

public int fibonacci(int num) {
    int fib = 1;
    if (num > 2)
        fib = fibonacci(num - 2) + fibonacci(num - 1);

    return fib;
}
```

Requisição Inicial

Requisição Recursiva

Métodos

Numa Classe além dos **atributos** podemos ter **métodos**.

Desta forma poderemos encapsular os atributos e controlar o acesso a estes através dos métodos.

Ou ainda, com os métodos podemos implementar mais funcionalidades a esta classe e seus objetos.

```
public class Contrato {
  private String numero;
  private String descr;
  private String titulo;

  public String getNumero() {
      return numero;
  }

  public void setNumero(String numero) {
      this.numero = numero;
  }

  public String getDescr() {
      return descr;
  }

  public void setDescr(String descr) {
      this.descr = descr;
  }

  public String getTitulo() {
      return titulo;
  }

  public void setTitulo(String titulo) {
      this.titulo = titulo;
  }
}
```

Encapsulamento

Uma Classe pode ser **publica** ou **privada**, **abstrata** ou **final**.

Public permite sua utilização em qualquer local, **private** oculta sua implementação.

Abstract exige a complementação se sua implementação, **final** impede sua extensão (herança).

```
private class Produto { }

public class Processo { }

public abstract class Figura {
    public abstract void exibir();
}

public final class String { }
```

Encapsulamento

Os atributos representam as informações que fazem parte das classes. Os métodos representam as ações ou mensagens que podemos requisitar ou enviar aos objetos.

Eles podem ser **privados**, **protegidos** e **públicos**.

A estas características permitem controlar o nível de encapsulamento destes atributos e métodos.

```
public abstract class Figura {
  private Integer coordenada;
  public String nome;
  protected String tipo;

  public void move(int x, int y) { }

  public abstract void desenha();
}
```

Modificadores

Podemos definir a forma de utilização dos atributos e métodos com os modificadores **static**, **final** e **transient**.

Com **static** fixamos os atributos e métodos na classe e não no objeto, com **final** criamos atributos constantes e impedimos a substituição de métodos, quando utilizando herança, por fim **transient** declara atributos temporários.

```
class Circulo extends Figura {
  private final Integer posicao = 100;
  private transient String temp;
  private static Integer tamanho;

  @Override
  public void desenha() { }
}
```

Controle de Fluxo

Através das estruturas de controle de fluxo de execução abaixo poderemos criar algoritmos para solucionar qualquer problema

- Seleção Simples
- Seleção Encadeada
- Seleção Composta
- Repetição com teste no início
- Repetição com teste ao final
- Repetição com variável de controle

Seleção Simples

Uma estrutura de seleção simples permite a escolha de um grupo de ações e estruturas a ser executado quando determinadas condições, representadas por expressões lógicas, são ou não satisfeitas

```
int idade = leInteiro("Informe sua Idade");

if(idade >= 18) {
    escrevaL("Você é maior de idade");
}
```

Seleção Composta

Na seleção composta, caso o resultado da condição seja falsa, teremos a execução de uma outra seqüência de comandos.

```
int idade = leInteiro("Informe sua Idade");

if(idade >= 18) {
    escrevaL("Você é maior de idade");
} else {
    escrevaL("Você é menor de idade");
}
```

Seleção Composta

Na seleção composta também permite que seja encadeado uma seqüência de testes

```
int idade = leInteiro("Informe sua Idade");

if(idade < 14) {
    escrevaL("Você ainda é uma criança");
} else if(idade < 18) {
    escrevaL("Você é quase maior de idade");
} else {
    escrevaL("Você é maior de idade");
}
```

Seleção Encadeada

Uma outra forma de tratar um problema de lógica com seleções encadeadas, quando a ação a ser executada depende do valor de uma variável

```
int codigo = leInteiro("Informe o código de acesso");

switch (codigo) {
case 1:
    escrevaL("Vá para o quinto andar");
    break;
case 3:
    escrevaL("Vá para o nono andar");
    break;
default:
    escrevaL("Vá para o primeiro andar");
    break;
}
```

Repetição com teste no início

Consiste numa estrutura de controle do fluxo lógico que permite executar diversas vezes um mesmo trecho do algoritmo, porém, sempre verificando antes de cada execução se é “permitido” repetir o mesmo trecho.

```
int quantidade = leInteiro("Informe a quantidade de valores");

double total = 0;
int contador = 1;
while (contador <= quantidade) {
    double valor = leReal("Informe o ", contador, "° valor");
    total = total + valor;
    contador = contador + 1;
}
escrevaL("O valor Total é de R$", total);
```

Repetição com teste ao final

Para realizar a repetição com teste no final, utilizamos a estrutura “repita”, que permite que um bloco ou ação primitiva seja repetida até que uma determinada condição seja verdadeira.

```
double total = 0;
int contador = 1;
char continua = 'N';
do {
    double valor = leReal("Informe o ", contador, "° valor");
    total = total + valor;
    contador = contador + 1;
    continua = leCaracter("Deseja continuar? (informe S ou N");
} while (continua == 'S');
escrevaL("O valor Total é de R$", total);
```

Repetição com variável de controle

A estrutura “para” repete a execução do bloco um número definido de vezes, pois ela possui limites fixos.

```
int quantidade = leInteiro("Informe a quantidade de valores");

double total = 0;
for(int contador = 1; contador <= quantidade; contador++) {
    double valor = leReal("Informe o ", contador, "º valor");
    total = total + valor;
}
escrevaL("O valor Total é de R$", total);
```

Vetores

- Simplifica a manipulação de diversas variáveis de um mesmo tipo de dado
- Armazenam um conjunto de itens que tenham o mesmo tipo de informação
- Os dados são acessados através de uma única variável
- São diferenciados e referenciados por um índice numérico

Vetores

- Estão presentes em praticamente todas as linguagens de programação
- Constituem um dos aspectos mais importantes e facilitadores no desenvolvimento de aplicações

Tipo da variável [ ] Nome da variável, Nome da variável ;

Ex.:

int [ ] notas ;

double [ ] [ ] vendasPorMes ;

Vetores

- Embora a declaração de um vetor seja idêntico ao de qualquer variável, é necessário que seja reservado a quantidade de itens que poderão ser armazenados no vetor antes de sua utilização

Ex.:

```
int [ ] notas = new int [ 5 ] ;

double [ ] [ ] vendasPorMes = new double [3] [12] ;
```

Vetores

- Para referenciarmos um vetor utilizamos o nome da variável, colchetes e uma literal ou variável que representa a posição dentro do vetor

- Esta posição não pode ultrapassar os limites inferior e posterior do vetor

```
int[ ] notas = new int[5];
notas[0] = 5;

int vendas = 0;
int mes = 0;
double[ ][ ] vendasPorMes = new double[3][12];
vendasPorMes[vendas][mes] = 345.93;
```

Estruturas de repetição

- Muitas vezes necessitamos de percorrer um vetor para efetuar uma operação
- Poderemos utilizar qualquer uma das três estruturas de repetição para efetuar qualquer operação.
- Normalmente utilizamos uma variável para controlar a posição no vetor
- Esta variável deve ser inicializada e incrementada a fim para que seja possível endereçar todos os itens do vetor

While

- A variável “i” representa o ponteiro do vetor

```
int[ ] notas = new int[5];

int i = 0;
while(i < notas.length) {
    notas[i] = leInteiro("Informe a ", i+1, "ª nota");
    i++;
}
```

Do…While

- Será necessário inverter o teste condicional ao se utilizar repita

```
int[ ] notas = new int[5];

int i = 0;
do {
    notas[i] = leInteiro("Informe a ", i+1, "ª nota");
    i++;
} while(i < notas.length);
```

For

- A estrutura de repetição “for” é a mais simples de se utilizar para estas operações

```
int[ ] notas = new int[5];

for(int i = 0; i < notas.length; i++) {
    notas[i] = leInteiro("Informe a ", i+1, "ª nota");
}
```

Vetor bidimensional

- Num vetor bidimensional necessitamos de dois índices para referenciarmos um elemento do vetor

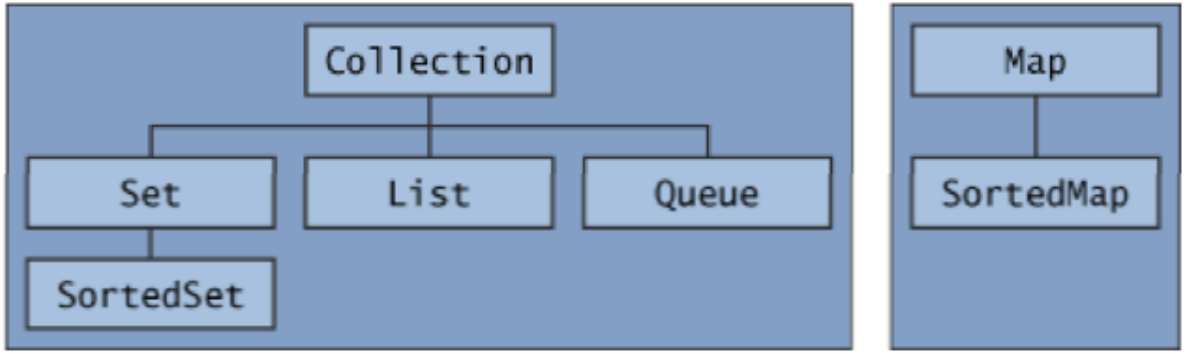
- Cada índice endereça uma dimensão no vetor

```
double[ ][ ] vendasPorMes = new double[3][12];

for (int i = 0; i < vendasPorMes.length; i++) {
    for (int j = 0; j < vendasPorMes[i].length; j++) {
        vendasPorMes[i][j] = leReal("Informe o ",i+1 ,
                "° valor de vendas para o ", j+1, "° mês");
    }
}
```

Coleções

O Java Collections Framework é a composição é uma arquitetura unificada para representar e manipular coleções e contém:

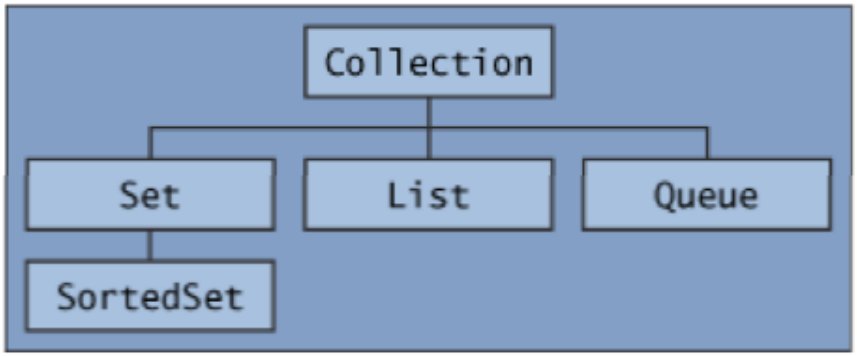
- Interfaces
- Implementações
- Algoritmos



Coleções

Uma Collection representa um grupo de objetos chamados de elementos.

A interface Collection é o último denominador comum para todas as implementações de coleções e é utilizada como argumento quando é necessário o máximo de generalização.

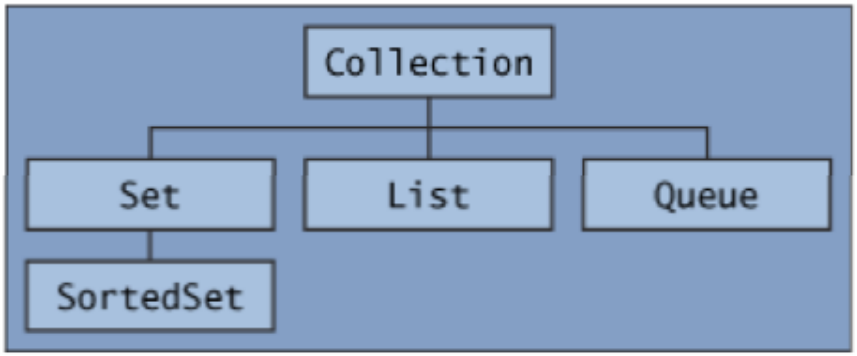


Coleções

Alguns tipo de coleções permitem duplicidade de elementos, e outros não.

Alguns são ordenados e outros não.

As mais específicas sub-interfaces são Set, List e Queue.



Collection

A Interface Collection tem vários métodos declarados e desta forma são comuns a todas as suas implementações, seja List, Set ou Queue.

Lista de alguns métodos de Collections

Métodos	Descrição
size()	Quantos elementos estão contidos na coleção
isEmpty()	true se a coleção estiver vazia
contains()	Verifica se um objeto está na coleção
add()	Adiciona um elemento à coleção
remove()	Remove um elemento à coleção
clear()	Remove todos os elementos de uma coleção
toArray()	Retorna um array dos elementos contidos na coleção
stream()	Retorna um objeto que suporta operações de agregação sequencial ou paralelo com os elementos da coleção

Set

O Set é uma coleção que não pode conter duplicidade de elementos.

Esta interface modela a abstração matemática de um set, tais como um conjunto de cartas em um jogo de truco, as disciplinas de um curso superior.

```
// Declara um set de Strings
Set<String> lista = new TreeSet<>();

// Inclui objetos no set
lista.add("abc");
lista.add("def");
lista.add("fgh");

// Ordena o set
// A implementação TreeSet ordena automaticamente na
// inserção e não permite duplicidade de chave

// Apresenta o set com foreach
for (String txt : lista)
    System.out.println(txt);

// Testa a existência do objeto no set
if(lista.contains("def"))
    // remove o objeto do set
    lista.remove("def");

// Apresenta o set com Stream
lista.stream().forEach(txt -> System.out.println(txt));
```

List

O List é uma coleção ordenada (as vezes chamada de seqüência).

List pode conter elementos duplicados.

Ao adicionarmos um elemento em List é associado um índice a este elemento (sua posição).

As implementações mais utilizadas de List são ArrayList e Vector.

```
// Declara uma lista de Strings
List<String> lista = new ArrayList<>();

// Inclui objetos na lista
lista.add("abc");
lista.add("def");
lista.add("fgh");

// Ordena a lista
Collections.sort(lista);

// Apresenta a lista com foreach
for (String txt : lista)
    System.out.println(txt);

// Testa a existência do objeto na lista
if(lista.contains("def"))
    // remove o objeto da lista
    lista.remove("def");

// Apresenta a lista com Stream
lista.stream().forEach(txt -> System.out.println(txt));
```

Queue

O Queue é uma coleção que armazena elementos em uma FIFO (primeiro que entra é o primeiro que sai).

Numa fila FIFO, todos os novos elementos são adicionados ao final da fila.

Métodos declarados em Queue

Métodos	Descrição
remove() e poll()	Remove e retorna o elemento no topo da fila
element() e peek()	Retorna o elemento no topo da fila sem removê-lo
offer()	Usado somente em Queues com limites, adiciona elementos

```java
// Declara uma Fila de Strings
Queue<String> lista = new PriorityQueue<>();

// Inclui objetos na Fila
lista.add("abc");
lista.add("def");
lista.add("fgh");

// Ordena a lista
// A implementação de PriorityQueue ordena
// automaticamente na inserção de objetos

// Apresenta a Fila com foreach
for (String txt : lista)
    System.out.println(txt);

// Testa a existência do objeto na Fila
if(lista.contains("def"))
    // remove o objeto da Fila
    lista.remove("def");

// Apresenta a Fila com Stream
lista.stream().forEach(txt -> System.out.println(txt));

// Retira os objetos da Fila
for (int i = lista.size(); i > 0; i--)
    System.out.println(lista.poll());
```

Map

O Map é um objeto que associa valores a chaves.

Um Map não pode conter chaves duplicadas.

Cada chave só pode estar associada a um valor.

Métodos declarados em Map

Métodos	Descrição
put()	Adiciona chaves e valores ao Map
get()	Retorna o elemento associado a chave informada
containsKey()	Retorna true se a chave existe no Map
containsValue()	Retorna true se o valor existe no Map
values()	Retorna uma Collection dos elementos contidos no Map

```java
// Declara um Mapa de Strings
Map<String, String> lista = new TreeMap<>();

// Inclui objetos no Mapa
lista.put("chave1", "abc");
lista.put("chave2", "def");
lista.put("chave3", "fgh");

// Ordena a lista
// A implementação de TreeMap ordena automaticamente
// na inserção de objetos pela chave e não permite
// duplicidade da chave e subistitui o valor caso
// isto aconteça

// Apresenta os valores do Mapa com foreach
for (String txt : lista.values())
    System.out.println(txt);

// Testa a existência do objeto no Mapa pela chave
if(lista.containsKey("chave2"))
    // remove o objeto do Mapa pela chave
    lista.remove("chave2");

// Apresenta os valores do Mapa com Stream
lista.values().stream().forEach(txt -> System.out.println(txt));
```

Exceções

try, catch e finally

- O termo **try** é utilizado para demarcar um bloco de código que pode gerar algum tipo de exceção
- O termo **catch** oferece um caminho alternativo a ser percorrido no case de ocorrer efetivamente uma exceção
- O termo **finally** delimita um bloco de código que será executado em quaisquer circunstâncias (ocorrendo ou não uma exceção)

Exceções

throw e throws

- A instrução **throw** é utilizada para gerar intencionalmente uma exceção

- O termo **throws** é utilizado para declarar um método que pode gerar uma exceção com a qual não consegue lidar

Exceções

A sintaxe geral de tratamento de exceções

```
try {
    // bloco
} catch(Exception1 var) {
    // bloco
    throw new Exception();
}
// ...
catch (Exception2 var) {
    // bloco
    throw new Exception();
} finally {
    // bloco
}
```

Exceções

ao lançar a **exceção**, é executado o **catch**

```
public static void main(String[] args) {
    int num = 0;
    while (num == 0) {
        try {
            String temp = JOptionPane.showInputDialog("Informe um N°");
            num = Integer.parseInt(temp);
            System.out.println(num);
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(null,"O n° informado é inválido!");
        }
    }
    System.out.println("Fim");
}
```