1 - Estrutura de um arquivo .java

```
//pacote - Opcional
br.com.empresa.teste;

// imports - Opcional
import java.utils.*;

// classes/interfaces - Opcional
class Pessoa {

    // atributos
    int a;

    // Construtores
    Pessoa() {}

    // métodos
    void x() {}
}
```

OBS.1: Comentários são opcionais e podem aparecer em qualquer lugar como:

```
// comentário de linha

/* comentário de bloco */

/** Padrão
*   Javadoc
*/
```

OBS.2: Todos esses elementos são opcionais, portanto, mesmo um arquivo vazio pode ser compilado.

OBS.3: Dentro de um construtor só é possível retornar com *return*, não é possível retornar alguma coisa, exemplo:

```
public myClass() {
    return null; // Erro
}

// ------------------------
public myClass() {
    return; // Válido
}
```

Sobre interfaces

OBS.4: Por padrão, variáveis de interfaces são públicas, estáticas e finais.

OBS.5: Por padrão, métodos de interfaces são públicos e abstratos.

OBS.6: Cada arquivo .java pode ter apenas uma classe pública, e esta classe deve ter o mesmo nome do arquivo.

2 - Pacotes java

OBS.1: Arquivos dentro do mesmo pacote não precisam de import para se referenciarem, a classe referenciada pode ou não ser pública.

OBS.2: Arquivos dentro de pacotes diferentes precisam de import para se referenciarem, e a classe referenciada deve ser pública.

OBS.3: Classes não públicas só podem ser acessadas dentro do pacote em que se encontram

OBS.4: Sejam duas classes, A e B, dentro do mesmo pacote, porém em arquivos separados. Se A faz uma referência a B, ao compilar a classe A, o compilador também irá compilar a classe B.

OBS.5: ***REGRAS - IMPORT DE CLASSES COM MESMO NOME:***

```
import teste1.A;
import teste2.A; //erro
//-------------------------------------------
import teste1.*;
import teste2.A;
...
A x = new A(); // x é do tipo teste2.A
//-------------------------------------------
import teste1.*;
import teste2.*;
...
A x = new A(); // erro (Qual A?)
//-------------------------------------------
teste1.A x = new teste1.A(); // x é do tipo teste1.A
teste2.A y = new teste2.A(); // y é do tipo teste2.A
//-------------------------------------------
teste1.A x = new teste2.A(); // erro de compilação
//-------------------------------------------
```

OBS.6: O comando *import pacote.*;* importa todos os TIPOS dentro do pacote especificado, portanto, este comando não importa tipos que estejam em sub pacotes dentro do pacote especificado.

OBS.7: O comando *import static nomePacote.nomeClasse.nomeEstatico* importa os itens (atributos e métodos) estáticos especificados da classe especificada. Assim, por exemplo, o comando *import static*

meupacote.minhaclasse. importa todos os itens estáticos da classe *minhaclasse*, mas NÃO se importa a classe *minhaclasse* em si.

OBS.8: ***NÃO É POSSÍVEL*** importar classes do pacote default, a única forma de referenciar um tipo do pacote default é em outro tipo que também esteja no pacote default. Por isso é tão recomendado separar os arquivos em pacotes.

3 - Executáveis Java

OBS.1: Para ser executável, a classe deve ter um método *main*, ele deve ser público e estático, além de possuir como argumento um array de strings (*String[] args ou String params[]*) ou um varargs de string (*String... args*).

```
public static void main(String... args) {} // Assinatura válida

public static void main(String[] args) {} // Assinatura válida

public static void main(String args[]) {} // Assinatura válida
```

OBS.2: Comando para compilar:

```
javac <nome_do_arquivo>.java
```

OBS.3: Comando para rodar:

```
java <nome_da_classe>
```

OBS.4: A classe não precisa ser pública para que se consiga rodar o método main, desde que a classe privada em que o método main se encontra possua o mesmo nome do arquivo.

OBS.5: O nome da classe, na verdade, é o nome do pacote concatenado com o nome da classe

OBS.6: é possível passar propriedades ao executar um *.class (-Dnomeparametro valor)* antes do nome da classe

```
java -Dpropriedade <nome_da_classe>
```

OBS.7: é possível passar parâmetros ao executar um *.class* após do nome da classe:

```
java <nome_da_classe> parametro1 parametro2
```

OBS.8: A variável *CLASSPATH* é uma variável de ambiente que indica onde o compilador deve procurar pelos arquivos .java ao compilar um programa. Ela pode ser alterada com uma flag (*-classpath ou -cp*) ao executar o comando *javac*.

```
javac -cp . <nome_do_arquivo>.java
```

OBS.9: O classpath, por padrão (variável de ambiente), é o diretório atual. Podemos adicionar novos diretórios ou arquivos *.jar* ao classpath utilizando ':' (para linux) ou ';' (para windows) para separar os diretórios:

```
-cp .:user/teste:programa.jar - Linux
-cp .;user/teste;programa.jar - Windows
```

OBS.10: Não é recomendado alterar o valor padrão da variável de ambiente *CLASSPATH*, e sim indicar o classpath ao compilar os arquivos usando as flags *-cp* e *-classpath*.

4 - Declaração e inicialização de variáveis

OBS.1: Não é possível utilizar variáveis que podem não ter sido inicializadas (variáveis locais não têm valor padrão).

OBS.2: São declaradas com tipo e nome (pelo menos) e valor.

VALORES PADRÃO

- Numéricos: 0
- Char: (Valor em branco, vide OBS.4)
- Boolean: false
- Referência: null

OBS.4: Char em java é considerado tipo numérico, assim, char padrão é o caractere correspondente ao valor 0 ("").

OBS.5: Apenas variáveis membro (globais) possuem valor padrão. Variáveis locais não possuem valor padrão e precisam ser inicializadas.

OBS.6: Arrays possuem valores padrão do tipo declarados para cada posição do array.

TIPOS PRIMITIVOS

- byte
- short
- char
- int
- long
- float
- double

- boolean

OBS.8: Boolean é o único tipo primitivo que não é numérico

OBS.9: float e double são tipos numéricos com ponto flutuante. Todos os demais tipos numéricos (byte, char, short, long, int) são numéricos

OBS.10: No java não existe o conceito de signed e unsigned, todos os tipos numéricos (exceto o char) vão de -x até (+x - 1).

OBS.11: O char é o único tipo numérico que não segue a OBS.10, indo de 0 até um valor positivo.

OBS.12: As variáveis numéricas com ponto flutuante (float e double) tamém podem assumir os seguintes valores:

- +infinito
- -infinito
- +0
- -0
- NaN

OBS.13: Valores literais só podem ser atribuídos a tipos primitivos. Exemplo:

```
int i = 15; // Válido
Carro carro = &5045; // Inválido (endereço de memória para uma referência de
objeto)
```

OBS.14: Valores literais de tipos numéricos são, por padrão int ou, caso haja um ponto, double. Exemplo:

```
int a = 10; // Internamente a é int
long a = 10; // Internamente a é int
float a = 10.0; // Internamente a é double
double a = 10.0; // Internamente a é double
```

OBS.15: Para mudar a regram da OBS.14 podemos usar as letras *l*, *f*, e *d*, maiúsculas ou minúsculas:

```
int a = 10; // Internamente a é int
long a = 10l; // Internamente a é long
float a = 10.0f; // Internamente a é float
double a = 10.0d; // Internamente a é double
```

OBS.16: Um número inteiro ao qual se atribui um valor literal iniciado com zero, é interpretado em base octal.

```
int a = 0761;
System.out.println(a) // Saída: 497
```

```
int b = 08;
System.out.println(b) // Erro, pois o símbolo 8 não existe na base octal
```

OBS.17: Um número inteiro ao qual se atribui um valor literal iniciado com "0X", é interpretado em base hexadecimal.

```
int a = 0XA;
System.out.println(a) // Saída: 10
```

OBS.18: Um número inteiro ao qual se atribui um valor literal iniciado com "0b", é interpretado em base binária.

```
int a = 0b10;
System.out.println(a) // Saída: 2

int b = 0b2;
System.out.println(b) // Erro, pois o símbolo 2 não existe na base binária
```

OBS.19: Um número com ponto flutuante pode ser escrito em notação científica da seguinte forma:

```
double a = 3.1E2; // A letra 'E' pode ser maiúscula ou minúscula
System.out.println(a) // Saída: 310.0, que equivale a 3.1x10^2
```

OBS.20: Para a *OBS.19* também vale a *OBS.15*, por padrão o valor é tratado internamente como double, para especificar que o valor deve ser float, utiliza-se a letra *f*.

```
float a = 3.1e2f; // Internamente a é float
```

OBS.21: É possível utilizar o caracter _ para separar visualmente os símbolos de valor de um número, desde que, antes e depois do caractere _ hajam símbolos que representem valor.

```
int a = 1_123_123;              // Válido, saída: 1123123
long b = 1_1234_1234l;          // Válido, saída: 112341234
long c = 1____1234_1234l;       // Válido, saída: 112341234
float d = 1_123.123_4f;         // Válido, saída: 1123.1234
float e = 0b10_0;               // Válido, saída 4
long f = _1_1234_1234l;         // Inválido, underline não está entre símbolos que
representam valores;
long g = 1_1234_1234_l;         // Inválido, underline não está entre símbolos que
representam valores;
float h = 1._2f;                // Inválido, underline não está entre símbolos que
representam valores;
```

```
float i = 1e_2f;                // Inválido, underline não está entre símbolos que
representam valores;
float j = 0_XAf;                // Inválido, underline não está entre símbolos que
representam valores;
long k = 0X_Af;                 // Inválido, underline não está entre símbolos que
representam valores;
...
```

OBS.22: Uma outra forma de se inicializar um char é através de seu caractere unicode, com aspas simples, barra invertida *u* ('\u') seguido do valor numérico do caractere segundo a tabela unicode.

```
char a = '\u03A9';
System.out.println(a) // Saída: Ω
```

NOMES DE VARIÁVEIS

OBS.23: Não é possível utilizar palavras reservadas para nomeação de variáveis, e nem os literais *true*, *false* e *null*.

OBS.24: Não é possível iniciar o nome da variável com um número, pois o número no início indica um valor literal.

OBS.25: É possível usar caracteres unicode no nome das variáveis.

```
int ã = 0;
System.out.println(ã); // Válido, saída: 0
```

OBS.26: Todos os nomes de variáveis são *case sensitive*.

5 - Variáveis de referência

Variáveis de referência vs Variáveis de tipos primitivos

Ao fazer uma atribuição a uma variável de tipo primitivo, a atribuição é feita por cópia.

```
int a = 10;
int b = a;
a += 5;
System.out.out.println(a); // 15
System.out.out.println(b); // 10
```

Ao fazer *b = a* é atribuído à variável *b* uma cópia do valor da variável *a*, assim, ao se alterar o valor de *a*, o valor de *b* permanece.

Já a atribuição a objetos não primitivos é feita por referência:

```
Obj a = new Obj();
a.nome = "nome";
Obj b = a;
a.nome = "nome2";
System.out.out.println(a); // nome2
System.out.out.println(b); // nome2
```

Nesse caso, *a* faz referência a uma posição de memória onde está armazenado um objeto do tipo *Obj*, ao fazer *b = a*, faz-se o objeto *b* apontar para a mesma posição que a variável *a*.

Dessa forma, ao se alterar os valores do objeto na posição especificada por *a*, *b* também apresentará os valores alterados.

Em resumo, há apenas um objeto *Obj* na memória, e duas variáveis que o referenciam, diferente dos tipos primitivos, onde cada variável possui, de fato, um valor.

RESUMO: Para variáveis de tipo primitivo as atribuições são feitas através da cópia de valores, para as variáveis de tipos não primitivos as atribuições são feitas a partir de referências de memória.

Ciclo de vida dos objetos

OBS.1: Um objeto só é criado ao se utilizar o operador *new* com o construtor do objeto.

```
Obj a;          // Objeto não foi criado ainda, apenas uma referência para um
objeto do tipo Obj
a = new Obj();  // Objeto foi criado, e uma referência a ele foi atribuída à
variável a
```

OBS.2: O componente responsável por remover da memória objetos que não são mais utilizados (referenciados) é chamado de *Garbage Collector*.

OBS.3: Um objeto são é removido da memória pelo *Garbage Collector* enquanto houver alguma referência a este objeto, ou seja, enquanto o objeto está acessível.

OBS.4: Ao se analizar a se um objeto ainda é referenciado ou não deve-se levar em conta o escopo das variáveis.

OBS.5: O escopo de uma variável é sempre o bloco na qual a variável foi declarada.

EXEMPLO

```
Carro a;            // Referência criada
a = new Carro();    // Objeto criado e atribuído à variável
a = new Carro();    // Novo objeto criado, aqui, o objeto criado anteriormente
passa a ser inacessível
```

```
Carro b;            // Referência criada

if(true){
    b = new Carro(); // Objeto criado e atribuído à variável
    Carro c = new Carro(); // Referência criada e Objeto criado e atribuído à
variável
}

/* Neste ponto do código, o objeto apontado pela variável b ainda é acessível,
pois b ainda está dentro de seu escopo e, portanto, ainda existe. */

/* Neste ponto do código, o objeto apontado pela variável c é inacessível, pois c
está de seu escopo (o escopo de c é o bloco if, vide OBS.5) e, portanto, não
existe. */
```

OBS.6: Quando um objeto torna-se inacessível ele passa a ser um objeto elegível a ser removido pelo *Garbage Collector*, porém, não é possível saber quando o objeto será removido. Assim, caso a questão pergunte quantos/quais objetos foram removidos da memória a resposta correta é que não é possível saber.

OBS.7: Referências indiretas também são válidas para considerar um objeto acessível.

```
Obj1 a = new Obj1();
Obj2 b = new Obj2();

b.obj1 = a;
a = null;

/* O objeto do tipo Obj1 criado inicialmente ainda é acessível, pois é
indiretamente referenciado por b.obj1 */
```

6 - Invocação de métodos

OBS.1:Métodos, assim como variáveis membro, podem ser invocados fora da classe a qual pertencem através do operador ponto '.', respeitando as regras de privacidade (*public, private, protected*).

```
class Obj2 () {
    ...
    Obj1 obj = new Obj1();
    obj.metodo();           // Válido
    obj.variavel;           // Válido

}
```

OBS.2:Dentro da classe, podem ser chamados sem o operador ponto, ou utilizando a palavra reservada *this*.

```
class Obj1 () {
    ...
    public void metodo() {}
    ...
    public void outroMetodo() {
        metodo();           // Válido
        this.metodo();      // Válido
    }
}
```

OBS.3:Se um método possui parâmetros, todos os parâmetros devem ser incluídos na chamada

```
public int metodo(int a, int b, int c);
...

metodo(0, 0, 0);    // Válido
metodo(0, 0);       // Erro
metodo();           // Erro
```

OBS.4:Se um método retorna um valor, não é obrigatório atribuir esse valor a uma variável.

```
public int metodo() {
    return 0;
}
...

int resultado = metodo();   // Válido
metodo();                   // Válido
```

Métodos com número variável de argumentos

OBS.5:Um método pode ter um número variável de argumentos utilizando *Varargs*.

Dessa forma, é possível enviar quantos parâmetros (do tipo especificado) forem necessários.

```
public void metodo(String... args);

metodo();                   // Válido
metodo("");                 // Válido
metodo("", "", "", "");     // Válido
```

OBS.6:Por mais que sejam parecidos, parâmetros *Varargs* e parâmetros *array* possuem diferenças em sua utilização.

```
public void metodoVarargs(String... args){}
public void metodoArray(String[] args){}
...
metodoArray("", "", "", "");        // Erro
metodoVarargs("", "", "", "");      // Válido

String[] strings = {"", "", "", ""};
metodoArray(strings);               // Válido
metodoVarargs(strings);             // Válido

// Em resumo, um método declarado com varargs aceita tanto vários argumentos como
um array do tipo especificado
// Todavia, um método declarado com array, só aceita array como argumento
```

OBS.7:Porém, dentro do método, a forma de acesso é a mesma para ambos os tipos.

```
public void metodoVarargs(String... args){
    System.out.println(args[0]);        // Válido
}

public void metodoArray(String[] args) {
    System.out.println(args[0]);        // Válido
}
```

OBS.8:Um parâmetro do tipo *Varargs* deve, obrigatoriamente, ser o último parâmetro na assinatura do método (isso não se aplica a parâmetros do tipo *array*).

```
public void metodo(String... args){}                    // Válido
public void metodo(int a, String... args){}             // Válido
public void metodo(int a, String... args, float b){}    // Erro
```

OBS.9: Não é possível enviar parâmetros de tipos diferentes, apenas do tipo especificado no método.

```
public void metodo(String... args){}
...
metodo("", 1, "");      // Erro
metodo("", "", "");     // Válido
```

7 - As classes StringBuffer e StringBuilder

São duas classes muito parecidas, que têm praticamente as mesmas funcionalidades.

OBS.1: A diferença entre elas é que a classe **StringBuilder** possui sincronização de métodos, permitindo que se use a mesma instância em threads diferentes. A classe **StringBuilder** foi introduzida no Java 5, já a classe

StringBuffer é mais antiga e está disponível em versões anteriores ao Java 5.

OBS.2: Ambas as classes servem para manipulação de Strings com tamanhos variáveis.

Utilização básica

```
StringBuffer sbf = new StringBuffer();          // cria um buffer vazio
System.out.println(sbf.toString());             // Saída:   (String vazia)

// Concatenando strings ao buffer
sbf.append("Hello");
sbf.append(" ");
sbf.append("World");
System.out.println(sbf.toString());             // Saída: Hello World

StringBuffer sbf2 = new StringBuffer(50);        /*  cria um buffer vazio
                                                    com tamanho 50  */
System.out.println(sbf2.toString());            // Saída:   (String vazia)

StringBuffer sbf3 = new StringBuffer(sbf);       // cria um buffer a partir de
outro
sbf3.append("!!!!");
System.out.println(sbf3.toString());            // Saída: Hello World!!!!
```

OBS.3: O mesmo vale para o **StringBuilder**:

```
StringBuilder sbd = new StringBuilder();         // cria um builder vazio
System.out.println(sbd.toString());             // Saída:   (String vazia)

// Concatenando strings com o builder
sbd.append("Hello");
sbd.append(" ");
sbd.append("World");
System.out.println(sbd.toString());             // Saída: Hello World

StringBuilder sbd2 = new StringBuilder(50);       /*  cria um builder vazio
                                                    com tamanho 50  */
System.out.println(sbd2.toString());            // Saída:   (String vazia)

StringBuilder sbd3 = new StringBuilder(sbd);       // cria um builder a partir de
outro
sbd3.append("!!!!");
System.out.println(sbd3.toString());            // Saída: Hello World!!!!
```

OBS.4: Também é possível instanciar um *buffer* ou *builder* passando uma string no construtor, assim, o objeto já é inicializado com um valor.

```
StringBuffer buffer = new StringBuffer("Hello"); // Válido
StringBuilder builder = new StringBuilder(" World"); // Válido
```

OBS.5: Caso um *buffer* ou *builder* seja declarado com um tamanho inicial específico, e a *string* ultrapasse o tamanho especificado, não ocorre erro, o próprio compilador trata o caso realocando mais memória para o objeto.

Métodos

append

```
StringBuilder builder = new StringBuilder();
StringBuilder builder2 = new StringBuilder(" World");
builder.append("Hello");            // Concatenar string
builder.append(builder2);           // Concatenar string a partir de outro builder
builder.append(new MeuObjeto());    /* Nesse caso, o valor concatenado é o
                                       retorno do método toString() da classe
                                       MeuObjeto */
```

insert

```
StringBuilder builder = new StringBuilder("Ho World");
builder.insert(1, "ell") ;       // Insere o texto especificado na posição
especificada
System.out.println(builder.toString());     // Saída: Hello World
```

delete

```
StringBuilder builder = new StringBuilder("Hello12345 World");
builder.delete(5, 10);                      /*/* Deleta o conteúdo entre as
posições
                                                especificadas */
System.out.println(builder.toString());     // Saída: Hello World
```

reverse

```
StringBuilder builder = new StringBuilder("Hello World");
builder.reverse();                      // Inverte o conteúdo da string
System.out.println(builder.toString());     // Saída: dlroW olleH
```

OBS.6: Também possuem todos os métodos da classe *String*;

OBS.7: Os métodos *subSequence* e *substring* das classes *StringBuilder* e *StringBuffer* não alteram o *Buffer/Builder*, apenas retornam os valores alterados por esses métodos.

```
StringBuilder builder = new StringBuilder("Hello World");
System.out.println(builder.toString());             // Saída: Hello World
System.out.println(builder.subSequence(1, 4));      // Saída: ell
System.out.println(builder.substring(1, 4));        // Saída: ell
System.out.println(builder.toString());             // Saída: Hello World
```

Criação de Strings

Formas de se criar uma string

Atribuição direta:

```
String a = "Teste";
```

Construtor:

```
String a = new String("Teste");
```

Construtor a partir de um array de *char*:

```
char[] array = new char[]{'T', 'e', 's', 't', 'e'};
String a = new String(array);
```

Construtor a partir de um *StringBuffer* ou *StringBuilder*:

```
StringBuffer sb = new StringBuffer("Teste");
String a = new String(sb);
```

Ao tentar imprimir algo que não seja uma string, o método *toString()* é chamado, até mesmo para um valor *null*.

```
System.out.println("Hello" + null); // Saída: Hello null
```

OBS.1: Deve-se tomar cuidado com a precedência de operadores ao tentar concatenar valores em uma string, pois o operador '+', serve para concatenação de strings, mas para números representa soma;

```
System.out.println("Teste" + 1 + 2);    // Saída: Teste12
System.out.println(1 + 2 + "Teste");    // Saída: 3Teste
```

8 - Manipulação de Strings

OBS.2: String são IMUTÁVEIS, geralmente as alterações geram novos objetos e mantém o objeto anterior inalterado;

```
String a = "teste";
a.toUpperCase();
System.out.println(a); // Saída: teste
```

Métodos importantes

Manipulação

- **String toUpperCase()** - transforma todos os caracteres em maiúsculo
- **String toLowerCase()** - transforma todos os caracteres em minúsculo
- **int length()** - retorna o tamanho da string (**length** para strings é um método, diferente dos **arrays**, onde é um atributo)
- **String charAt(int c)** - retorna o caractere na posição especificada (atenção com os limites da string, risco de **StringIndexOutofBoundsException**)
- **boolean isEmpty()** - Retorna **true** se a string é vazia (“”.isEmpty() - true ; “ “.isEmpty() - false ; null.isEmpty() - Erro, **NullPointerException**)
- **String trim()** - retira espaços em branco no começo e no fim
- **String substring(int beginIndex, int endIndex)** - devolve a substring a partir dos índices de começo e fim;
- **String substring(int beginIndex)** - semelhante ao anterior, mas toma a substring a partir do índice passado até o final da String;
- **String concat(String)** - concatena o parâmetro ao fim da String atual e devolve o resultado;
- **String replace(char oldChar, char newChar)** - substitui todas as ocorrências de determinado char por outro;
- **String replace(CharSequence target, CharSequence replacement)** - substitui todas as ocorrências de determinada CharSequence (como String) por outra.

Comparação

- **boolean equals(Object)** - compara igualdade caractere a caractere (herdado de Object);
- **boolean equalsIgnoreCase(String)** - compara caractere a caractere ignorando maiúsculas/minúsculas;
- **int compareTo(String)** - compara as 2 Strings por ordem lexicográfica (vem de Comparable);
- **int compareToIgnoreCase(String)** - compara as 2 Strings por ordem lexicográfica ignorando maiúsculas/minúsculas.

Busca

- **boolean contains(CharSequence)** - devolve true se a String contém a sequência de chars;
- **boolean startsWith(String)** - devolve true se começa com a String do parâmetro;
- **boolean endsWith(String)** - devolve true se termina com a String do parâmetro;
- **int indexOf(char) e int indexOf(String)** - devolve o índice da primeira ocorrência do parâmetro;
- **int lastIndexOf(char) e int lastIndexOf(String)** - devolve o índice da última ocorrência do parâmetro.

9 - Uso de operadores

Para valores literais e tipos primitivos

O operador de atribuição é o '='.

OBS.1: Só é possível fazer uma atribuição caso o valor especificado (no lado direito do operador '=') seja um valor menos abrangente que o tipo especificado no lado esquerdo do operador '='. Por exemplo:

```
long a = 15; // Válido - O valor 15 "cabe" dentro de um long
byte b = 999999; // ERRO -  O valor 9999999 não "cabe" dentro de um byte, ou seja, long é um tipo mais abrangente que byte
double d = 30.0f; // Válido - Também é válido, mesmo com o modificador 'f', pois um float "cabe" em um double.
```

Lembre-se da ordem de abrangência dos tipos numéricos primitivos:

byte < short < int < long < float < double

Onde *float* e *double* são números de ponto flutuante, e os demais tipos inteiros.

OBS.2: Lembre-se de tomar cuidado com o tipo *char* na regra anterior, pois *char* é um tipo numérico que só assume valores positivos (possui a mesma abrangência que o *short*).

Justamente pelo fato de o *char* não assumir valores negativos, apesar de possuírem a mesma abrangência, é impossível fazer uma atribuição de *char* para *short* e nem de *short* para *char*.

```
short a = 1;
char c = a; // ERRO
```

Ao atribuir um valor literal para um *short*, *byte* ou *char* avalia-se em tempo de compilação se o valor literal atribuído é um valor válido para aquele tipo.

Resumo:

byte < short < int < long < float < double

- short ou acima = byte // OK
- int ou acima = short // OK
- long = int // OK
- float ou double = int // OK
- float ou double = long // OK
- double = float = double // OK
- short = char // NOK
- char = short // NOK

Para variáveis de referência

Sejam uma classe A, e uma classe B que extende a classe A, pode-se considerar a classe A mais abrangente que a classe B, assim:

```
B b = new B();
A a = b;            // Válido

B c = a;            // Inválido
```

OBS.3: Lembrando que na atribuição de variáveis de referência, o valor atribuído é a referência para o objeto em memória, e não uma cópia do objeto.

Operadores aritméticos

- + Soma
- - Subtração
- * Multiplicação
- / Divisão
- % Resto de divisão

O retorno de uma operação aritmética segue duas regras gerais:

- O valor retornado é, no mínimo um *int* (nunca um *short*, *char* ou *byte*), assim, mesmo que se esteja somando dois *bytes* (por exemplo), o retorno será um *int*
- O tipo do valor retornado será o mesmo do maior argumento da operação (por exemplo, seja uma soma entre um *int* e um *float*, o retorno será um *float*)

Divisão por zero

Uma divisão com inteiros resulta em erro de compilação, já se o retorno for um ponto flutuante, o resultado pode ser mais ou menos infinito

```
double a = 5.0/0;   // Válido, resultado: Infinito
double b = -5/0.0;  // Válido, resultado: -Infinito
double c = 5/0;     // Erro, inteiro dividido por zero
```

Operadores de comparação

- == igual
- != diferente
- < menor
- > maior
- <= menor ou igual
- >= maior ou igual

Para valores literais e tipos primitivos

Ao se comparar duas variáveis de diferentes tipos numéricos, o compilador ignora o tipo e compara apenas o valor, assim:

```
int a = 2;
float b = 2;
a == b; // True, pois ambos valem 2
```

OBS.4: Para bollean, os únicos operadores válidos são '==' e '!=';

OBS.5: Ao tentar comparar um tipo numérico com um não numérico (bollean ou referência) ocorre erro de compilação. O mesmo ocorre na comparação entre bollean e variável de referência. Ou seja, a comparação deve fazer sentido para aqueles valores.

Operadores lógicos

Os operadores lógicos existentes no java são os seguintes:

- & (and, ou operador 'e')
- | (or, ou operador 'ou')
- ^ (xor, ou operador 'ou exclusivo')

Por padrão, ao usar esses operadores o java processa ambas as partes da expressão, ou seja:

```
(1!=1 & 2==2) // tanto a expressão 1!=1 como a 2==2 serão verificadas, mesmo que a
primeira já torne o resultado falso
```

A fim de economizar processamentonessas verificações, existe o recurso *short circuit*, para, caso a primeira verificação já determine o resultado, as verificações seguintes nem são executadas.

Para utilizar o *short circuit* deve-se escrever o operador em questão duas vezes:

```
(1!=1 && 2==2)  // apenas a primeira verificação será executada, resultado: false
(1!=1 & 2==2)   // ambas as verificações serão executadas, resultado: false

(1==1 || 2!=2)  // apenas a primeira verificação será executada, resultado: true
(1==1 | 2!=2)   // ambas as verificações serão executadas, resultado: true
```

O *short circuit* não se aplica ao operador '^', pois para uma operação de ou exclusivo é sempre necessário validar todas as expressões.

```
(1==1 ^ 2!=2)       // Válido, resultado: true
(1==1 ^^ 2!=2)      // Não compila
```

Operadores de incremento e decremento

- ++, incrementa uma unidade
- --, decrementa uma unidade

Podem ser pré atribuição ou pós atribuição

- i++, retorna o valor de i e em seguida incrementa a variável
- ++i, incrementa a variável e em seguida retorna o valor de i

Sendo assim:

```
int i = 5;
System.out.println(i++);    // Imprime 5, pois a atribuição vem antes
System.out.println(i);      // Imprime 6

System.out.println(++i);    // Imprime 7, pois incremento vem antes
```

Também podem ser usados junto com o operador de atribuição para incremento e decrementa de mais unidades.

```
a += 1;     // equivale a: a = a + 1
a -= 1;     // equivale a: a = a - 1
a *= 2;     // equivale a: a = a * 2
a /= 2;     // equivale a: a = a / 2
a %= 2;     // equivale a: a = a % 2
```

Em caso de problemas de compatibilidade de tipos ao usar essas atribuições acima, o compilador não acusa erro de compilação, o programador deve tomar cuidado com as questões de precisão:

```
byte a = 1;
a += 10000;                 // Não ocorrerá erro, mas o resultado não será como esperado, pois 10001 não cabe em um byte
System.out.println(a);      // imprime 21
```

Operador ternário

Composto por uma condição, um retorno para caso a condição seja true e um retorno para caso a condição seja false. Segue a seguinte estrutura:

(condição) ? (retorno true) : (retorno false)

```
int a = (1 == 1) ? 1 : 2;
System.out.println(a);      // imprime 1
```

É importante observar o tipo da variável q recebe a atribuição e os tipos dos retornos:

```
int a = (1 == 1) ? "1" : 2;     // não compila, pois 'a' espera um inteiro, e um dos possíveis retornos é string
```

Outros operadores

- operador de referência: (.)
- operador de concatenação de strings: (+)

O operador de referência serve para acessar recursos de um objeto (respeitando as regras de visibilidade)

```
obj.nome;           // Acessamos o atributo nome do objeto obj
obj.toString();     // Acessamos o método toString do objeto obj
```

O operador + serve concatenar strings (quando aplicado entre strings)

```
System.out.println("Hello" + "World");      // Imprime 'Hello World'
```