

Sintaxe

2.1 - Sintaxe e Semântica

Tudo na vida possui forma e conteúdo, mas um bom conteúdo com forma ruim é um problema, e uma forma bacana com conteúdo ruim é pior ainda.

Projetar software é lidar com isso toda hora. Modelos, especificações, código, e precisam de uma boa forma (sintaxe) e um bom conteúdo (semântica).

Entender como ter qualidade na sintaxe e na semântica é fundamental.

Quanto à semântica, de um modo geral é o estudo do significado das coisas (do conteúdo das “formas”). Na Engenharia de Software, quando falamos de semântica nos referimos ao significado dos modelos, ao nível de entendimento (clareza, objetividade, detalhamento, coesão, etc.) de algo.

Por exemplo: uma redação bem escrita (sem erros de concordância, gramática, etc.), mas com um texto ruim (sem contexto, sem concordância, com uma história confusa, sem clareza e fluidez) possui boa “sintaxe”, mas uma “semântica” ruim.

2.1.1 - Anatomia e Escopo

Em Ciência da Computação escopo é um contexto delimitante aos quais valores e expressões estão associados. Linguagens de programação têm diversos tipos de escopos. O tipo de escopo vai determinar quais tipos de entidades este pode conter e como estas são afetadas, em outras palavras, a sua semântica.

Normalmente, o escopo é utilizado para definir o grau de ocultação da informação, isto é, a visibilidade e acessibilidade às variáveis em diferentes partes do programa.

Escopos podem:

- Conter declarações ou definições de identificadores;
- Conter instruções e/ou expressões, que definem um algoritmo executável ou parte dele;
- Aninhar ou ser aninhados.

Informação

Em resumo, escopo é a acessibilidade de objetos, variáveis e funções em diferentes partes do código.

A grande maioria das linguagens oferecem uma estrutura bem definida de codificação, seguindo o pseudocódigo conforme abaixo:

java

```
1  class {
2
3      //escopo global ou de classe
4      globalNumber = 10
5
6      methodOne(){
7          //escopo de método
8          myNumber = 1
9
10         print(globalNumber) //10
11     }
12
13     methodTwo(){
14         //escopo de método
15         myNumber = 1
16
17         //escopo de bloco ou de fluxo
18         if(true){
19
20             yourNumber = 2
21
22             myNumber = yourNumber
23         }
24
25         print(myNumber) //2
26         print(yourNumber) //error - acesso somente dentro do fluxo
27
28     }
29
30 }
```

```
50 | }
```

Exemplo

Uma parte fundamental na elaboração de algoritmos simples ou complexos é determinar a localização do código em questão. Sem um domínio sobre escopo de códigos, seu projeto tende a conter falhas estruturais e comprometer a proposta principal da aplicação.

java

```
1  public class Conta {
2      //variavel da classe conta
3      double saldo=10.0;
4
5      public void sacar(Double valor) {
6          //variavel do método
7          double novoSaldo = saldo - valor;
8      }
9      public void imprimirSaldo(){
10         //disponível em toda classe
11         System.out.println(saldo);
12         //somente o método sacar conhece esta variavel
13         System.out.println(novoSaldo);
14
15     }
16 }
```

Sucesso

O exemplo ilustrado acima segue as diretrizes recomendadas e aplicadas em qualquer projeto acadêmico ou corporativo. A seguir você terá mais esclarecimentos destas regras e convenções de escrita.

2.1.1.1 Anatomia das classes

Um dos principais recursos que utilizamos no paradigma de orientação a objetos são as classes, com elas podemos abstrair e criar representações do mundo real para o código.

```
1 public class MinhaClasse {  
2  
3     //SEU CÓDIGO AQUI  
4  
5 }
```

- 99,9% das nossas classes iniciarão com `public class` .
- Toda classe precisa de nome, exemplo `MinhaClasse` .
- Após o nome, precisamos definir o corpo `{ }` onde iremos compor nossas classes com atributos e métodos.

```
1 public class MinhaClasse {  
2     // CORPO DA CLASSE  
3  
4     public static void main(String[] args) {  
5         // CORPO DO MÉTODO MAIN  
6     }  
7  
8 }
```

- É de suma importância, que agora você consiga se localizar, dentro do conjunto de chaves `{ }` existentes em sua classe.
- Dentro de uma aplicação, **recomenda-se que somente uma classe possua o método `main`** , responsável por iniciar todo o nosso programa.
- O método `main` recebe seu nome `main`, sempre terá a visibilidade `public`, será definido como `static`, não retornará nenhum valor com `void` ,e receberá um parâmetro do tipo array de caracteres `String[]` .

Padrão de nomenclatura

Quando se trata de escrever códigos na linguagem Java, é recomendado seguir algumas convenções de escrita. Esses padrões estão expressos nos itens abaixo:

- Arquivo `.java`: Todo arquivo `.java` deve começar com letra MAIÚSCULA. Se a palavra for composta, a segunda palavra deve também ser maiúscula, exemplo:

`Calculadora.java` , `CalculadoraCientifica.java`

- Nome da classe no arquivo: A classe deve possuir o mesmo nome do arquivo `.java`, exemplo:

```
1 // arquivo CalculadoraCientifica.java
2
3 public class CalculadoraCientifica {
4
5 }
```

2.1.1.2 Variáveis e métodos

Como identificar a diferença entre, declaração de variáveis e métodos em nossa programação? Existe uma estrutura comum para ambas as finalidades, exemplo:

```
1 // Estrutura
2 Tipo NomeBemDefinido = Atribuicao (opcional em alguns casos)
3
4 // Exemplo
5
6 int idade = 23;
7 double altura = 1.62;
8 Dog spike; //observe que aqui a variável spike não tem valor, é normal
```

Uma variável é um meio de armazenar dados em memória. Em outras palavras, ela funciona como um container para valores em um programa

Nome de variável: toda variável deve ser escrita com letra minúscula, porém se a palavra for composta, a primeira letra da segunda palavra deverá ser MAIÚSCULA, exemplo:

ano e anoFabricacao . O nome dessa prática para nomear variáveis dessa forma se chama **camelCase**.

Informação

Existe uma regra adicional para variáveis, quando na mesma queremos identificar que ela não sofrerá alteração de valor, exemplo: queremos determinar que uma variável de nome br sempre representará "Brasil" e nunca mudará seu valor, logo, determinamos como escrita o código abaixo:

```
1 String BR = "Brasil"
2 double PI = 3.14
3
```

```
4 | int ESTADOS_BRASILEIRO = 27
   | int ANO_2000 = 2000
```

🚨 Cuidado

Recomendações: Para declarar uma variável nós podemos utilizar caracteres, números e símbolos, porém, devemos seguir algumas regras da linguagem.

- Deve conter apenas letras, _ (underline), \$ ou os números de 0 a 9 ;
- Deve obrigatoriamente se iniciar por uma letra (preferencialmente), _ ou \$, jamais com número;
- Deve iniciar com uma letra minúscula (boa prática – ver abaixo);
- Não pode conter espaços;
- Não podemos usar palavras-chave da linguagem;
- O nome deve ser único dentro de um escopo.

java

```
1 | // Declaração inválida de variáveis
2 |
3 | int numeroUm = 1; //Os únicos símbolos permitidos são _ e $
4 | int 1numero = 1; //Uma variável não pode começar com numérico
5 | int numero um = 1; //Não pode ter espaço no nome da variável
6 | int long = 1; //long faz parte das palavras reservadas da linguagem
7 |
8 | // Declaração válida de variáveis
9 | int numero$um = 1;
10 | int numero1 = 1;
11 | int numeroum = 1; ou numeroUm
12 | int longo = 1;
```

- Estrutura de declaração de métodos:

As mesmas regras estruturais de variáveis se aplicam a métodos, em breve iremos abordar mais convenções e estruturação de métodos.

java

```
1 | // Estrutura
2 | TipoRetorno NomeObjetivoNoInfinitivo Parametro(s)
3 |
4 | //Exemplo
5 |
6 | int somar (int numeroUm, int numero2)
```

🔔 Atenção

Como parte da estrutura de declaração de variáveis e métodos, também temos o aspecto da **visibilidade**, mas ainda não é necessário nesta etapa de estudos.

2.1.1.3 Indentação

Basicamente **indentar** é um termo utilizado para escrever o código do programa de forma hierárquica, facilitando assim a visualização e o entendimento do programa.



```
if places.count > 0 {
    for i in 0..

```

Abaixo, veja um exemplo de um algoritmo de validação, de aprovação de estudante.

► Sem indentação:

► Com indentação:

2.1.1.4 Hora da verdade

Que tal colocar em prática tudo o que apresentamos agora?

Tente criar uma classe Conta Corrente contendo os atributos e métodos listados abaixo.

Hora da verdade

Usando um bloco de notas crie uma classe denominada Conta Corrente com os atributos e métodos.

Atributos	Métodos
✓ Número da conta	✓ Sacar um valor
✓ Número da agência	✓ Transferir um valor para outra conta
✓ Nome do cliente	✓ Cancelar a conta com uma justificativa
✓ Data de nascimento	✓ Consultar o extrato entre duas datas
✓ Saldo da conta	✓ Consultar o saldo atual



2022 // Digital

Sucesso

O ponto mais relevante, em compreender a definição dos tipos de dados é o momento da definição do tipo para uma variável. Se precisar de mais uma ajudinha, consulte o [Link](#) em nosso canal .

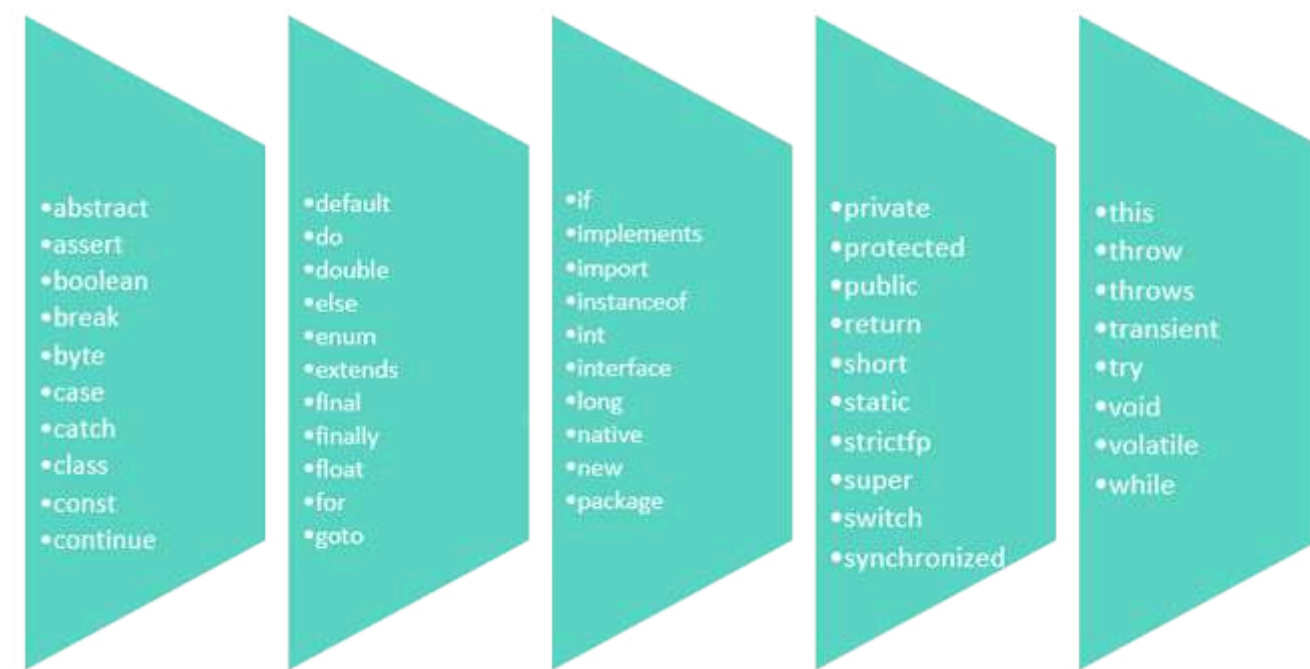
2.1.2 - Palavras Reservadas

Palavras reservadas do java, ou keywords(palavras-chave), são palavras que possuem significado específico no código.

A linguagem contém 52 palavras distribuídas em categorias com finalidades e fronteiras de uso bem definidas conforme tabela abaixo:

2.1.2.1 Descrevendo as palavras

As palavras reservadas na linguagem Java possuem regras de indentificação e utilização conforme descrição abaixo:



- Todas as palavras reservadas são minúsculas
- Nenhuma palavra reservada poderá ser utilizada definir nome de variáveis e métodos
- As palavras reservas determinam recursos ou características de forma individual ou agrupada para o seu algoritmo
- Conhecer e compreender todas as 52 palavras reservadas leva tempo e prática

2.1.2.2 Agrupando as palavras

Para uma melhor interpretação e compreensão do uso das palavras reservadas em nosso cotidiano, é necessário primeiramente, uma organização e classificação das mesmas. Sendo assim, será mais fácil compreender sua aplicabilidade

Modificadores de acesso

Palavra	Descrição	Classe	Variável	Método
public	Acesso de qualquer classe	X	X	X

Palavra	Descrição	Classe	Variável	Método
private	Acesso apenas dentro da classe	X	X	X
protected	Acesso por classes no mesmo pacote e subclasses	X	X	X

Tipos primitivos

Palavra	Descrição	Classe	Variável	Método
boolean	um valor indicando verdadeiro ou falso		X	no retorno
byte	um inteiro de 8 bits (signed)		X	no retorno
char	um carácter Unicode (16bit unsigned) (signed)		X	no retorno
double	um carácter Unicode (16bit unsigned) (signed)		X	no retorno
float	um número de ponto flutuante de 32 bits		X	no retorno

Modificadores de classes, variáveis ou métodos

Palavra	Descrição	Classe	Variável	Método
abstract	classe que não pode ser instanciada ou método que precisa ser implementado, por uma subclasse não abstrata	X		
class	especifica uma classe	X		
extends	indica a superclasse que a subclasse está estendendo	X		

Palavra	Descrição	Classe	Variável	Método
final	impossibilita que uma classe seja estendida, que um método seja sobrescrito ou que uma variável seja reinicializada	X	X	X
implements	indica as interfaces que uma classe irá implementar	X		
interface	especifica uma interface	X		
native	indica que um método está escrito em uma linguagem dependente de plataforma, como o C			X
new	instancia um novo objeto, chamando seu construtor		X	
static	faz um método ou variável pertencer à classe ao invés de às instâncias		X	
strictfp	usado em frente a um método ou classe para indicar que os números de ponto flutuante seguirão as regras de ponto flutuante, em todas as expressões			X
synchronized	indica que um método só pode ser acessado por uma thread de cada vez			X
transient	impede a serialização de campos		X	
volatile	indica que uma variável pode ser alterada durante o uso de threads		X	

Controle de fluxo dentro de um bloco de código

Palavra	Descrição	Classe	Variável	Método
break	sai do bloco de código em que ele está			fluxo
case	executa um bloco de código dependendo do teste do switch			X
continue	pula a execução do código que viria, após essa linha e vai para a próxima passagem do loop			fluxo
default	executa esse bloco de código caso nenhum dos teste de switch-case seja verdadeiro			X
do	executa um bloco de código uma vez, e então realiza um teste em conjunto com o while para determinar se o bloco deverá ser executado novamente			X
else	executa um bloco de código alternativo caso o teste "if" seja falso			X
for	usado para realizar um loop condicional de um bloco de código			X
if	usado para realizar um teste lógico de verdadeiro ou falso			X
instanceof	determina se um objeto é uma instância de determinada classe, superclasse ou interface			X
return	retorna um método sem executar qualquer código, que venha			X

Palavra	Descrição	Classe	Variável	Método
	depois desta linha (também pode retornar uma variável)			
switch	indica a variável a ser comparada nas expressões case			X
while	executa um bloco de código repetidamente enquanto a condição for verdadeira			X

Tratamento de erros

Palavra	Descrição	Classe	Variável	Método
assert	testa uma expressão condicional, para verificar uma suposição do programador			X
catch	declara o bloco de código usado para tratar uma exceção			X
finally	bloco de código, após um try-catch, que é executado independentemente do fluxo de programa seguido ao lidar com uma exceção			X
throw	usado para passar uma exceção para o método que o chamou			X
throws	indica que um método pode passar uma exceção para o método que o chamou			assinatura
try	bloco de código que tentará ser executado, mas que pode causar uma exceção			X

Controle de pacotes

Palavra	Descrição	Classe	Variável	Método
import	importa pacotes ou classes para dentro do código	X		
package	especifica a que pacote, todas as classes de um arquivo pertencem.	X		

Variáveis de referência

Palavra	Descrição	Classe	Variável	Método
super	refere-se a superclasse imediata			X
this	refere-se a instância atual do objeto			X

Palavras reservadas não utilizadas

Palavra	Descrição	Classe	Variável	Método
const	não utilize para declarar constantes			
goto	não implementada na linguagem Java, por ser considerada prejudicial			

2.1.2.3 Combinação de palavras

Abaixo, iremos exercitar algumas das possibilidades mais recorrentes em combinar o uso das palavras reservadas a níveis de: Classe, Atributos e Métodos

Nível	Combinação	Explicação
classe	public class	Determina que a classe é pública
classe	abstract class	Determina que a classe é abstrata

Nível	Combinação	Explicação
classe	final class	Determina que a classe é final e não pode ser herdada
classe	class A extends B	Determina que a classe A herda da classe B
classe	class A implements B	Determina que a classe A implementa a interface B
atributo	public static	Determina que o atributo é público e estático (nível de classe)
atributo	public static final	Determina que o atributo é público, estático e inalterável (constante)
método	abstract void	Determina que o método é abstrato e sem retorno
método	synchronized void	Determina que o método é sincronizado e sem retorno

Atenção

A tabela acima é somente uma ilustração e um direcionamento para você poder fixar um pouco mais o conceito das palavras reservadas na linguagem Java.

2.1.2.4 Palavras "opostas"

Assim como nas classificações gramaticais da língua portuguesa, existem algumas palavras que são completamente opostas (antônimas) na linguagem Java conforme tabela abaixo:

Palavra	Palavra	Explicação
package	import	Enquanto package determina o diretório real da classe, o import informa de onde será importada a classe. Isso porque, podemos ter classes de mesmo nome.

Palavra	Palavra	Explicação
extends	implements	enquanto extends determina que uma classe estende outra classe, implements determina que uma classe implementa uma interface, porém nunca o contrário.
final	abstract	enquanto final determina fim de alteração de valor ou lógica comportamental, abstract em métodos, exige que sub-classes precisarão definir comportamento e um método abstrato. NOTA: Se uma classe contém um único método abstrato, toda classe precisa ser.
throws	throw	Esta é uma das situações mais complicadas, de compreensão destas duas palavras. Enquanto a throws determina que um método pode lançar uma exceção, throw é a implementação que dispara a exceção. Vamos conhecer mais sobre este conceito no assunto Exceções.

2.1.3 - Tipos e Variáveis

Com toda certeza, uma hora ou outra, seja na sua vida acadêmica ou profissional na área de desenvolvimento você já ouviu falar em tipos de variáveis. Isso se deve ao fato de que o computador tem a necessidade que o programador “explique” para ele o que exatamente quer, da forma mais especificada possível, e no Java não é diferente.

2.1.3.1 Tipos primitivos

Em Java, existem palavras reservadas especiais para representar tipos básicos de dados que são essenciais para construir programas. Estes tipos básicos são chamados de tipos primitivos.

Para fixar

Os oito tipos primitivos em Java são:

`int` , `byte` , `short` , `long` , `float` , `double` , `boolean` e `char` – esses tipos não são objetos e portanto representam valores brutos. Eles são armazenados diretamente na pilha de memória.

Tabela de Tipos Primitivos e seus valores:

Tipo	Memória	Valor Mínimo	Valor Máximo
byte	1 byte	-128	127
short	2 byte	-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

Os tipos primitivos, que podem conter partes fracionárias:

Tipo	Memória	Valor Mínimo	Valor Máximo	Precisão
float	4 bytes	-3,4028E + 38	3,4028E + 38	6 – 7 dígitos
double	8 bytes	-1,7976E + 308	1,7976E + 308	15 dígitos

Embora o tipo `float` ocupe menos espaço na memória do que o tipo `double` , ele é menos utilizado devido a uma limitação na precisão decimal entre 6 e 7 dígitos.

Com os avanços nos computadores, não há mais a necessidade de se preocupar com o uso dos tipos `short` e `byte` , pois a memória é abundante.

De maneira semelhante, o tipo `long` também é pouco utilizado, pois valores grandes são raros de se trabalhar.

Portanto, na maioria das situações, utilizamos o tipo `int` para representar números inteiros ou `double` para representar números fracionados.

Informação

Devemos compreender que os tipos primitivos sempre terão um valor padrão mesmo NÃO havendo uma atribuição explícita, veja o código abaixo:

```
1 byte    b; //b=0
2 short   s; //s=0
3 int     i; //i=0
4 long    l; //l=0
5
6 float   f; //f=0.0
7 double  d; //d=0.0
8
9 char    c; //c='\u0000'
10 boolean o; //b=false
```

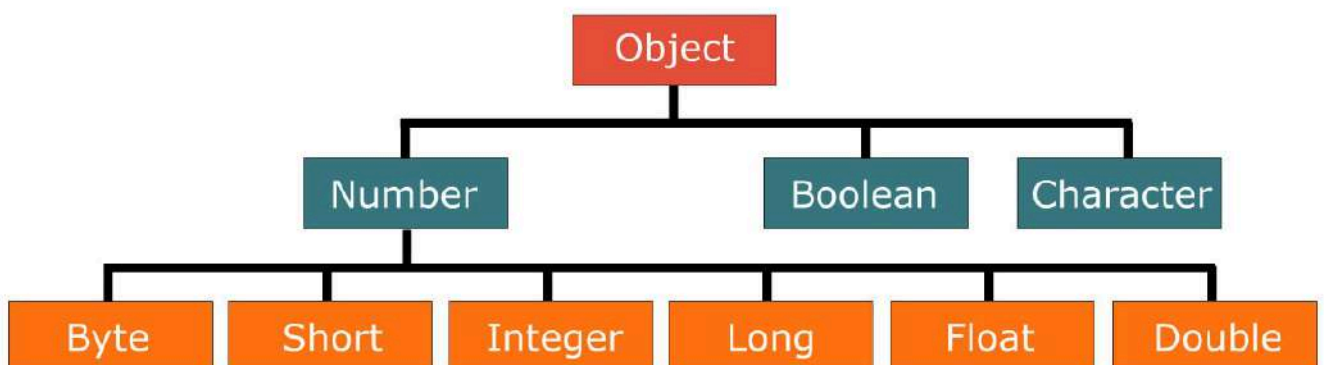
java

2.1.3.2 Tipos Wrappers

Os tipos primitivos não são objetos, mas às vezes é necessário trabalhar com eles como se fossem. Por exemplo, quando você precisa passar um valor primitivo como um parâmetro de um método, você precisa converter o valor primitivo em um objeto. Nesse caso, você pode usar os tipos wrappers.

Os **tipos wrappers** são classes que representam os tipos primitivos. Eles são imutáveis, o que significa que uma vez criado um objeto wrapper, seu valor não pode ser alterado para garantir a segurança.

Wrapper Class Hierarchy



Os seguintes tipos Wrappers estão disponíveis em Java:

```
1 Integer    (int)
2 Long       (long)
3
```

java

4	Float	(float)
5	Double	(double)
6	Short	(short)
7	Byte	(byte)
8	Boolean	(boolean)
	Character	(char)

Conversão de Tipos

A conversão de tipos é necessária quando você deseja armazenar um valor de um tipo de dado em uma variável de outro tipo. Por exemplo, você pode armazenar um valor `int` em uma variável de tipo `double` porque o tipo `double` pode armazenar valores inteiros e fracionários.

Conversão de Tipos Explícita: é feita por meio de uma expressão de conversão de tipos. Ela é necessária quando você deseja converter um valor de um tipo de dado em outro que não seja compatível com o dado original.

Conversão de Tipos Implícita: é feita automaticamente pelo compilador da linguagem não sendo necessário nenhum tipo de demarcação.

```
1 // conversão de tipos explicita
2 double d = 100.0;
3 int i = (double) d;
4 System.out.println(i); // 100
5
6 // conversão de tipos implícita
7 double d = 100.0;
8 int i = d;
9 System.out.println(i); // 100
```

java

? O que aconteceria

```
1 // Tentar converter um double de valor 3.75 para um int ?
2 double d = 3.75;
3 int i = (double) d;
4 System.out.println(i); // ???
```

java

Métodos Wrapper

Os tipos Wrappers em Java oferecem uma série de métodos úteis para a manipulação de tipos primitivos. Alguns desses métodos incluem:

- **Métodos de parsing:** Esses métodos permitem a conversão de strings em tipos primitivos equivalentes. Por exemplo, o método `parseInt()` pode ser usado para converter uma string em um inteiro, e o método `parseDouble()` pode ser usado para converter uma string em um número de ponto flutuante. Esses métodos são úteis quando você precisa ler dados de uma fonte externa e convertê-los em tipos primitivos.

java

```
1 String num = "100";
2 int i = Integer.parseInt(num);
3 System.out.println(i); // 100
```

- **Métodos de conversão:** Esses métodos permitem a conversão de tipos primitivos para objetos e vice-versa. Por exemplo, o método `valueOf()` pode ser usado para converter um tipo primitivo em um objeto equivalente, enquanto o método `intValue()` pode ser usado para converter um objeto `Integer` em um inteiro.

java

```
1 String num = "100";
2 Integer obj = Integer.valueOf(n);
3 System.out.println(obj); // 100
4
5 Integer obj = new Integer(100);
6 int i = obj.intValue();
7 System.out.println(i); // 100
```

🔴 Para fixar

Em Java muitos métodos possuem uma estrutura denominada de **sobrecarga**. Isso quer dizer que, métodos que possuem o mesmo nome, mesmo retorno, porém parâmetros diferentes.

java

```
1 Integer.valueOf(int i)
2 Integer.valueOf(String s)
```

- **Métodos de comparação:** Esses métodos permitem comparar objetos Wrappers para verificar se são iguais ou se um é maior ou menor que o outro. Por exemplo, o método `equals()` pode ser usado para comparar dois objetos `Integer` para verificar se eles são iguais, enquanto o método `compareTo()` pode ser usado para comparar dois objetos de forma mais geral.

```
1 Integer obj1 = new Integer(100);
2 Integer obj2 = new Integer(100);
3 System.out.println(obj1.equals(obj2)); // true
4 System.out.println(obj1.compareTo(obj2)); // 0
```

- Métodos de informação: Esses métodos fornecem informações sobre o tipo Wrapper, como o valor mínimo e máximo que pode ser representado. Por exemplo, o método `maxValue()` pode ser usado para obter o valor máximo representável para um tipo Wrapper específico, enquanto o método `minValue()` pode ser usado para obter o valor mínimo representável.

```
1 System.out.println(Integer.MAX_VALUE); // 2147483647
2 System.out.println(Integer.MIN_VALUE); // -2147483648
```

2.1.3.3 Tipos customizados

Os tipos customizados são bibliotecas ou **pacotes externos** que não fazem parte por padrão no Java, mas são adicionados ao projeto para fornecer recursos adicionais que podem ser úteis na construção de aplicativos. Então você pode usar esses tipos customizados em seu código para adicionar funcionalidades extras.

Alguns exemplos de tipos customizados incluem:

Nome do Projeto	Pacote principal	Finalidade
Joda Time	org.joda.time	Fornecer soluções mais avançadas de datas e horas incluindo suporte a fusos horários e operações de data/hora mais avançadas.
Gson	com.google.gson	Fornecer suporte para converter objetos Java em sua representação JSON e vice-versa.
Jackson	com.fasterxml.jackson	Fornecer suporte para processamento de JSON, incluindo conversão de objetos

Nome do Projeto	Pacote principal	Finalidade
		Java para e de JSON.
Apache Commons	org.apache.commons	Fornecer uma série de utilitários que podem ser usados para tarefas comuns, como manipulação de strings, arquivos e coleções.
Apache POI	org.apache.poi	Fornecer suporte para leitura e escrita de arquivos do Microsoft Office, incluindo arquivos do Excel.
Apache Log4j	org.apache.log4j	Fornecer suporte para registro de eventos.
Hibernate	org.hibernate	Fornecer suporte para mapeamento objeto-relacional (ORM).
Junit	org.junit	Fornecer suporte para testes unitários.

Por que usar bibliotecas externas?

Elas fornecem uma série de recursos úteis que ajudam a simplificar o desenvolvimento de aplicações Java, economizando tempo e esforço para o desenvolvedor, e essas bibliotecas são amplamente utilizadas na comunidade Java e são bem documentadas, o que significa que há uma ampla comunidade de desenvolvedores que as usam e podem ajudar em caso de dúvidas ou problemas.

Como usar bibliotecas externas?

Para usar bibliotecas externas em seu projeto, você precisa adicioná-las ao seu projeto. Existem várias maneiras de fazer isso, mas a maneira mais comum é usar o gerenciador de dependências do Maven. O Maven é um gerenciador de dependências que permite gerenciar as dependências de um projeto Java, incluindo bibliotecas externas.

Como adicionar bibliotecas externas ao seu projeto?

Existem várias maneiras de adicionar bibliotecas externas ao seu projeto, mas a maneira mais comum é usar o gerenciador de dependências do Maven. O Maven é um

gerenciador de dependências que permite gerenciar as dependências de um projeto Java, incluindo bibliotecas externas.

```
1 <dependency>
2   <groupId>com.google.gson</groupId>
3   <artifactId>gson</artifactId>
4   <version>2.8.6</version>
5 </dependency>
```

xml

Como usar bibliotecas externas em seu código?

Depois de adicionar uma biblioteca externa ao seu projeto, você pode usá-la em seu código. Por exemplo, se você adicionar a biblioteca Gson ao seu projeto, você pode usar a classe Gson para converter objetos Java em sua representação JSON e vice-versa.

```
1 import com.google.gson.Gson;
2
3 public class Main {
4     public static void main(String[] args) {
5         Gson gson = new Gson();
6         String json = gson.toJson(new int[] { 1, 2, 3, 4, 5 });
7         System.out.println(json); // [1,2,3,4,5]
8     }
9 }
```

java

2.1.4 - Declaração e Atribuição

Declaração e atribuição de variáveis é uma das etapas fundamentais em programação, pois permite armazenar dados na memória do computador para uso futuro. A aplicação de regras claras e precisas ajuda a evitar erros de sintaxe e de lógica, bem como torna o código mais legível e fácil de manter.

Os métodos com e sem retorno também são importantes, pois permitem organizar o código em blocos reutilizáveis e mantê-lo claro e conciso. Os métodos com retorno devolvem um valor, enquanto os métodos sem retorno não retornam nada. É importante escolher a abordagem adequada para cada tarefa.

Finalmente, tornar uma variável uma constante pode ser útil quando você quer garantir que seu valor não mude acidentalmente durante a execução do programa. Isso ajuda a preservar a integridade dos dados e a evitar erros de lógica. Em geral, é uma boa prática usar constantes sempre que possível para garantir que o código seja claro, preciso e fácil de manter.

2.1.4.1 Declaração de variáveis

Uma variável é uma referência a um espaço de memória utilizado pelo seu programa. De acordo com as convenções da linguagem de programação, cada variável é composta por três elementos: tipo de dados, identificação e valor atribuído.

A estrutura padrão para se declarar uma variável sempre é:

```
<Tipo> <nomeVariável> <atribuiçãoDeValorOpcional>
```

Exemplos abaixo:

```
1  int idade; //Tipo "int", nome "idade", com nenhum valor atribuído.
2  int anoFabricacao = 2021; //tipo "int", nome "anoFabricacao", com valor 2021.
3  double salarioMinimo = 2.500; //tipo "double", nome "salarioMinimo", valor 2.500
```

java

Atenção

Existe algumas peculiaridades a trabalhar com alguns tipos específicos. Observe no exemplo abaixo:

```
1  public class TipoDados {
2  public static void main(String[] args) {
3      byte idade = 123;
4      short ano = 2021;
5      int cep = 21070333; // se começar com 0, talvez tenha que ser outro tipo
6      long cpf = 98765432109L; // se começar com 0, talvez tenha que ser outro tipo
7      float pi = 3.14F;
8      double salario = 1275.33;
9  }
10
```

java

}

✓ Conclusão

Observe que o tipo `long` precisa terminar com `L`, o tipo `float` precisa terminar com `F` e alguns cenários do dia-a-dia, podem estimular uma alteração de tipos de dados convencional.

Muitas das vezes criamos uma variável, definimos um valor correspondente, manipulamos esta variável e temos consciência de seu valor na aplicação. Mas, cuidado!

🔔 Atenção

Java é linguagem de programação fortemente "tipado".

Veja o exemplo abaixo:

java

```
1 // TiposEVariaveis.java
2
3 short numeroCurto = 1;
4 int numeroNormal = numeroCurto;
5 short numeroCurto2 = numeroNormal;
6
7 // Mesmo sabendo que numeroNormal é igual a numeroCurto,
8 // não é possível atribuir a numeroCurto2
```

2.1.4.2 Declaração de constantes

As **Constantes**, são valores armazenados em memória que não podem ser modificados depois de declarados. Em Java, esses valores são representados pela palavra reservada `final`, seguida do tipo. Por convenção, **Constantes** são sempre escritas em CAIXA ALTA.

Abaixo, temos um exemplo explicativo sobre uso de variáveis e constantes:

java

```
1 public class ExemploVariavel {
2     public static void main(String[] args) {
3         /*
```

```

4      * esta linha é considerada como declaração de variável iniciamos a existência
5      * variável numero com valor 5 regra: tipo + nome + valor
6      */
7      int numero = 5;
8
9
10     /*
11     * na linha abaixo iremos alterar o valor do variável para 10 observe que o tip
12     * não é mais necessário, pois a variável já foi declarada anteriormente
13     */
14     numero = 10;
15
16     System.out.print(numero);
17
18     /*
19     * ao usar a palavra reservada final, você determina que jamais
20     * esta variavel poderá obter outro valor;
21     * logo a linha 25 vai apresentar um erro de compilação
22     * isso é considerado uma CONSTANTE na linguagem Java
23     */
24     final double VALOR_DE_PI = 3.14;
25
26     VALOR_DE_PI=3.15; //Esta linha vai apresentar erro de compilação!
27 }

```

Atenção

Compreendemos que, para declarar uma variável como uma constante, utilizamos a palavra **final** , mas por convenção, esta variável deverá ser escrita toda em caixa alta.

2.1.4.3 Declaração de métodos

Uma classe é definida por atributos e métodos. Já vimos que atributos são, em sua grande maioria, variáveis de diferentes tipos e valores. Os métodos, por sua vez, correspondem a funções ou sub-rotinas disponíveis dentro de nossas classes.

Critério de Nomeação de Métodos

Esses critérios não são obrigatórios, mas é recomendável que sejam seguidos, pois essas convenções facilitam a vida dos programadores ao trabalharem em códigos de forma colaborativa. Ao seguir estas convenções, tornamos o código mais legível para nós e também para outras pessoas. Para métodos, os critérios são:

- Deve ser nomeado como verbo;
- Seguir o padrão camelCase (Todas as letras minúsculas com a exceção da primeira letra da segunda palavra).

Exemplo de nomeação de métodos:

```
1  somar(int n1, int n2){}
2
3  abrirConexao(){ }
4
5  concluirProcessamento() {}
6
7  findById(int id){}
8
9  calcularImprimir(){}
```

java

2.1.4.4 Definindo parâmetros e retorno de métodos

Mas, como sabemos a melhor forma, de definir os métodos das nossas classes? Para chegar à essa conclusão, somos auxiliados por uma convenção estrutural para todos os métodos. Essa convenção é determinada pelos aspectos abaixo:

1. **Qual a proposta principal do método?** Você deve se perguntar constantemente até compreender a real finalidade do mesmo.
2. **Qual o tipo de retorno esperado após executar o método?** Você deve analisar se o método será responsável por retornar algum valor ou não.
3. **Qual o tipo de parâmetro esperado pelo método?** Você deve analisar se o método irá receber algum parâmetro ou não.
4. **O método possui o risco de apresentar alguma exceção?** Exceções são comuns na execução de métodos, as vezes é necessário prever e tratar a possível existência de uma exceção.
5. **Qual a visibilidade do método?** Avaliar se será necessário que o método seja visível a toda aplicação, somente em pacotes, através de herança ou somente a nível a própria

classe.

Abaixo, temos um exemplo de uma classe com alguns métodos e suas respectivas considerações:

java

```
1  public class MyClass {
2
3      public double somar(int num1, int num2){
4          //LOGICA - FINALIDADE DO MÉTODO
5          return ... ;
6      }
7
8      public void imprimir(String texto){
9          //LOGICA - FINALIDADE DO MÉTODO
10         //AQUI NÃO PRECISA DO RETURN
11         //POIS NÃO SERÁ RETORNADO NENHUM RESULTADO
12     }
13     // throws Exception : indica que o método ao ser utilizado
14     // poderá gerar uma exceção
15     public double dividir(int dividendo, int divisor) throws Exception{}
16
17     // este método não pode ser visto por outras classes no projeto
18     private void metodoPrivado(){}
19
20     //alguns equívocos estruturais
21     public void validar(){
22         //este método deveria retornar algum valor
23         //no caso boolean (true ou false)
24     }
25     public void calcularEnviar(){
26         //um método deve representar uma única responsabilidade
27     }
28     public void gravarCliente(String nome, String cpf, Integer telefone, ....){
29         //este método tem a finalidade de gravar
30         //informações de um cliente, por que não criar
31         //um objeto cliente e passar como parâmetro ?
32         //veja abaixo
33     }
34     public void gravarCliente(Cliente cliente){}
35     //ou
36     public void gravar(Cliente cliente){}
37
```

2.1.5 - Operadores

Você já ouviu aquela frase antiga que diz que programação é simplesmente uma lista de instruções lógicas que, quando executadas, nos entrega resultados incríveis? Bem, é isso mesmo! Quando começamos a escrever o nosso primeiro código, logo percebemos que uma das coisas mais comuns é pedir informações ao usuário, fazer algumas operações lógicas e, em seguida, apresentar o resultado final.



Classificação do Operadores

2.1.5.1 Atribuição

Representado pelo símbolo de igualdade = .

O operador de atribuição é utilizado para definir o valor inicial ou sobrescrever o valor de uma variável. em Java, definimos um tipo, nome e opcionalmente atribuímos um valor à variável através do operador de atribuição. Exemplos abaixo:

```
1 //classe Operadores.java
2 String nome = "GLEYSON";
3 int idade = 22;
4 double peso = 68.5;
5 char sexo = 'M';
6 boolean doadorOrgao = false;
7 Date dataNascimento = new Date();
```

2.1.5.2 Aritméticos

O operador aritmético, é utilizado para realizar operações matemáticas entre valores numéricos, podendo se tornar ou não uma expressão mais complexa.

Os operadores aritméticos são: + (adição), - (subtração), * (multiplicação), / (divisão) e % (módulo).

```
1 //classe Operadores.java
2 double soma = 10.5 + 15.7;
3 int subtração = 113 - 25;
4 int multiplicacao = 20 * 7;
5 int divisao = 15 / 3;
6 int modulo = 18 % 3;
7 double resultado = (10 * 7) + (20/4);
```

Atribuição abreviada de aritméticos

Acabamos de aprender sobre operadores de atribuição e aritméticos, porém a linguagem Java oferece meios de abreviação juntando estes dois contextos conforme abaixo:

```
1 //classe Operadores.java
2 /*
3     Vamos imaginar que n1 começaria com valor 10 e n2 com valor 5
4     mas em seguida gostaria de somar o valor de n1 e n2 e atribuir a n2.
5 */
6 int n1 = 10;
7 int n2 = 5;
8
9
```

```

10 //forma literal
11 n2 = n2 + n1;
12
13 //forma abreviada
14 n2 += n1;
15
16 System.out.println(n2);

```

🔔 Atenção

O operador de adição (+), quando utilizado em variáveis do tipo texto, realizará a “concatenação de textos”.

```

1 //classe Operadores.java
2 String nomeCompleto = "LINGUAGEM" + "JAVA";
3
4 //qual o resultado das expressões abaixo?
5 String concatenacao = "?";
6
7 concatenacao = 1+1+1+"1";
8
9 concatenacao = 1+"1"+1+1;
10
11 concatenacao = 1+"1"+1+"1";
12
13 concatenacao = "1"+1+1+1;
14
15 concatenacao = "1"+(1+1+1);

```

java

2.1.5.3 Relacionais

Os operadores relacionais, avaliam a relação entre duas variáveis ou expressões. Neste caso, mais precisamente, definem se o operando à esquerda é igual, diferente, menor, menor ou igual, maior, maior ou igual ao da direita, retornando um valor booleano como resultado.

- == Quando desejamos verificar se uma variável é IGUAL A outra.
- != Quando desejamos verificar se uma variável é DIFERENTE da outra.

- > Quando desejamos verificar se uma variável é MAIOR QUE a outra.
- >= Quando desejamos verificar se uma variável é MAIOR OU IGUAL a outra.
- < Quando desejamos verificar se uma variável é MENOR QUE outra.
- <= Quando desejamos verificar se uma variável é MENOR OU IGUAL a outra.

java

```
1 //classe Operadores.java
2 int numero1 = 1;
3 int numero2 = 2;
4
5 if(numero1 > numero2)
6     System.out.print("Numero 1 maior que numero 2");
7
8 if(numero1 < numero2)
9     System.out.print("Numero 1 menor que numero 2");
10
11 if(numero1 >= numero2)
12     System.out.print("Numero 1 maior ou igual que numero 2");
13
14 if(numero1 <= numero2)
15     System.out.print("Numero 1 menor ou igual que numero 2");
16
17 if(numero1 != numero2)
18     System.out.print("Numero 1 é diferente de numero 2");
```

2.1.5.4 Unários

Esses operadores, são aplicados juntamente com um outro operador aritmético. Eles realizam alguns trabalhos básicos como incrementar, decrementar, inverter valores numéricos e booleanos.

- (+) Operador unário de valor positivo – números são positivos sem esse operador explicitamente;
- (-) Operador unário de valor negativo – nega um número ou expressão aritmética;
- (++) Operador unário de incremento de valor – incrementa o valor em 1 unidade;
- (--) Operador unário de decremento de valor – decrementa o valor em 1 unidade;
- (!) Operador unário lógico de negação – nega o valor de uma expressão booleana.

Exemplos abaixo:


```
1 //classe Operadores.java
2 int numero = +5; //ou somente 5
3
4 //Imprimindo o numero negativo
5 System.out.println(- numero);
6
7 //incrementando numero em mais 1 numero, imprime 6
8 numero ++;
9 System.out.println(numero);
10
11 //incrementando numero em mais 1 numero, imprime 7
12 System.out.println(numero ++); // ops algo de errado não está certo
13
14 System.out.println(numero); // agora sim, numero virou 7
15
16 //ordem de precedência conta aqui
17 System.out.println(++ numero);
18
19 boolean verdadeiro = true;
20
21 System.out.println("Inverteu " + !verdadeiro);
```

Cuidado

Muito cuidado com ordem de precedência, dos operadores unários!

2.1.5.5 Ternário

O operador de condição ternária é uma forma resumida para definir uma condição e escolher por um dentre dois valores. Você deve pensar numa condição ternária, como se fosse uma condição IF normal, porém, de uma forma em que toda a sua estrutura estará escrita numa única linha.

O operador ternário é representado pelos símbolos "?" e ":" utilizados na seguinte estrutura de sintaxe:

<Expressão Condicional> ? <Caso condição seja true> : <Caso condição seja false>

Atenção

O operador ternário muitas das vezes é interpretado como um controle de fluxo, sendo que o seu papel principal é uma atribuição condicionada.

Exemplo abaixo:

java

```
1 // classe Operadores.java
2 int a, b;
3
4 a = 5;
5 b = 6;
6
7 /* EXEMPLO DE CONDICIONAL UTILIZANDO UMA ESTRUTURA IF/ELSE
8 if(a==b)
9     resultado = "verdadeiro";
10 else
11     resultado = "falso";
12 */
13
14 //MESMA CONDICIONAL, MAS DESSA VEZ, UTILIZANDO O OPERADOR CONDICIONAL TERNÁRIO
15 String resultado = (a==b) ? "verdadeiro" : "falso";
16
17 System.out.println(resultado);
```

2.1.5.6 Lógicos

Os operadores lógicos, representam o recurso que nos permite criar expressões lógicas maiores, a partir da junção de duas ou mais expressões.

- && - Operador Lógico "E"
- || - Operador Lógico "OU"

Java

```
1 // Operadores.java
2 boolean condicao1=true;
3
4 boolean condicao2=false;
5
6 /* Aqui estamos utilizando o operador lógico E para fazer a união de duas
7 expressões.
```

```

8      É como se estivesse escrito:
9      "Se Condicao1 e Condicao2 forem verdadeiras, executar código"
10     */
11
12     if(condicao1 && condicao2)
13         System.out.print("Os dois valores precisam ser verdadeiros");
14
15     //Se condicao1 OU condicao2 for verdadeira, executar código.
16     if(condicao1 || condicao2)
17         System.out.print("Um dos valores precisa ser verdadeiro");

```

Expressões Lógicas Avançadas

Nós acabamos de aprender que existem os operadores lógicos `&` (E) e `||` (OU), mas por que no exemplo acima, foram ilustradas as condições:

```

1      if (condicao1 && condicao2) e if(condicao1 || condicao2) ?

```

✓ Conclusão

A duplicidade nos operadores lógicos é um recurso conhecido como Operador Abreviado, isso quer dizer que, se a condição1 atender aos critérios, não será necessário validar a condição2.

```

1      // ComparacaoAvancada.java
2      int numero1 = 1;
3      int numero2 = 1;
4
5      if(numero1== 2 & numero2 ++ == 2 )
6          System.out.println("As 2 condições são verdadeiras");
7
8      System.out.println("O numero 1 agora é " + numero1);
9      System.out.println("O numero 2 agora é " + numero2);
10
11     // Vamos mudar a linha 5 do código acima para: if(numero1== 2 && numero2 ++ == 2

```

java

✓ Conclusão

O mesmo acontece com o operador `|` e `||`, considerando que operador, agora representa que se uma das alternativas for verdadeira, a outra nem precisa ser avaliada.

2.1.6 - Comentários

Comentários, como o próprio nome instiga, são notas que podem ser incluídas no código fonte para descrever o que se quiser. Assim, não modificam o programa executado e servem somente para ajudar o programador a melhor organizar os seus códigos. Os comentários em Java seguem a mesma sintaxe da linguagem C++.

2.1.6.1 Linha única

Para representar uma linha de comentário utilizamos o prefixo `//` e assim todo o conteúdo deixará de ser interpretado pelo processo de checagem e compilação.

```
1  {
2      // esta linha é um comentário e abaixo parte de nosso algoritmo
3      int a=10;
4  }
```

java

Outro exemplo

```
1  {
2      // este é um bloco de programação
3      int a=10;
4      int b=1;
5      if (b==3) {
6          // este é um bloco que é executado se b for igual a 3
7          b=a*10;
8      } else {
9          // este é um bloco que é executado se b for diferente de 3
10         int a=100;
11         b=a*10;
12     }
13     System.out.println("O valor de b é " + b);
14 }
```

java

2.1.6.2 Múltiplas linhas

As vezes nosso comentário precisará ser um pouco mais esclarecedor necessitando assim mais de uma linha. Neste cenário, utilizamos os símbolos `/* */` para esta finalidade.

java

```
1  {
2      int a=10;
3      int b;
4      b=a*2;
5
6      /* a partir deste ponto, deve-se começar a exibir
7      os resultados na tela do usuário */
8  }
```

Cuidado

Não utilize comentários com o propósito de descrever um algoritmo mal estruturado, busque sempre seguir as convenções e boas práticas em desenvolvimento

Abaixo iremos demonstrar o mesmo algoritmo primeiro sem usar das boas práticas e em seguida uma refatoração mais legível.

► **Código complexo:**

► **Código normalizado:**

2.1.6.3 Javadoc

Javadoc é um gerador de documentação criado pela Sun Microsystems para documentar a API dos programas em Java, a partir do código-fonte. O resultado é expresso em HTML. É constituído, basicamente, por algumas marcações muito simples inseridas nos comentários do programa.

Este sistema é o padrão de documentação de classes em Java, e muitas dos IDEs desta linguagem irão automaticamente gerar um Javadoc em HTML.

Tags

Os desenvolvedores usam certos estilos de comentários e tags Javadoc ao documentar códigos-fonte. Um bloco de comentário em Java iniciado com `/**` irá iniciar um bloco de comentário Javadoc, que será incluído no HTML gerado. Uma tag Javadoc começa com um "@" (arroba). Na tabela abaixo, algumas destas tags.

tag	descrição
@author	Nome do desenvolvedor
@deprecated	Marca o método como deprecated. Algumas IDEs exibirão um alerta de compilação se o método for chamado.
@exception	Documenta uma exceção lançada por um método — veja também @throws.
@param	Define um parâmetro do método. Requerido para cada parâmetro.
@return	Documenta o valor de retorno. Essa tag não deve ser usada para construtores ou métodos definidos com o tipo de retorno void.
@see	Documenta uma associação a outro método ou classe.
@since	Documenta quando o método foi adicionado a a classe.
@throws	Documenta uma exceção lançada por um método. É um sinônimo para a @exception introduzida no Javadoc 1.2.
@version	Exibe o número da versão de uma classe ou um método.

Vamos explorar como documentar um classe simples que realizar a divisão entre dois números inteiros.

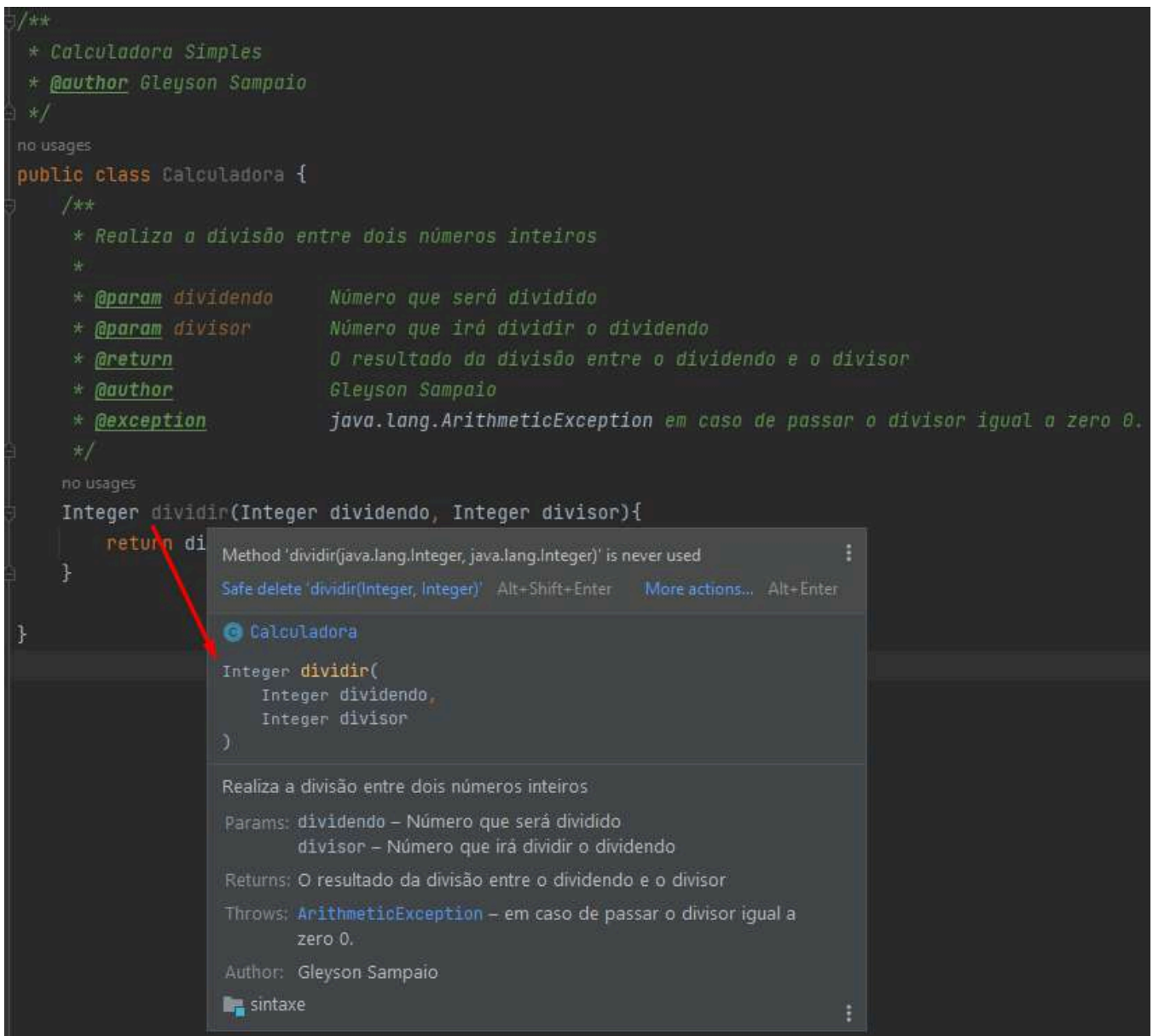
java

```
1  /**
2   * Calculadora Simples
3   * @author Gleyson Sampaio
4   */
5  public class Calculadora {
6      /**
7       * Realiza a divisão entre dois números inteiros
8       *
9       * @param dividendo      Número que será dividido
```

```

10      * @param divisor      Número que irá dividir o dividendo
11      * @return              O resultado da divisão entre o dividendo e o divisor
12      * @author              Gleyson Sampaio
13      * @exception           java.lang.ArithmeticException em caso de passar o di
14      */
15      static Integer dividir(Integer dividendo, Integer divisor){
16          return dividendo / divisor;
17      }
18  }

```



2.1.7 - Java Beans

Um das maiores dificuldades na programação é escrever algoritmos legíveis, a níveis que sejam compreendidos por todo seu time ou por você mesmo no futuro. Para isso, a linguagem Java sugere através de convenções, formas de escrita universal, para nossas classes, atributos, métodos e pacotes.

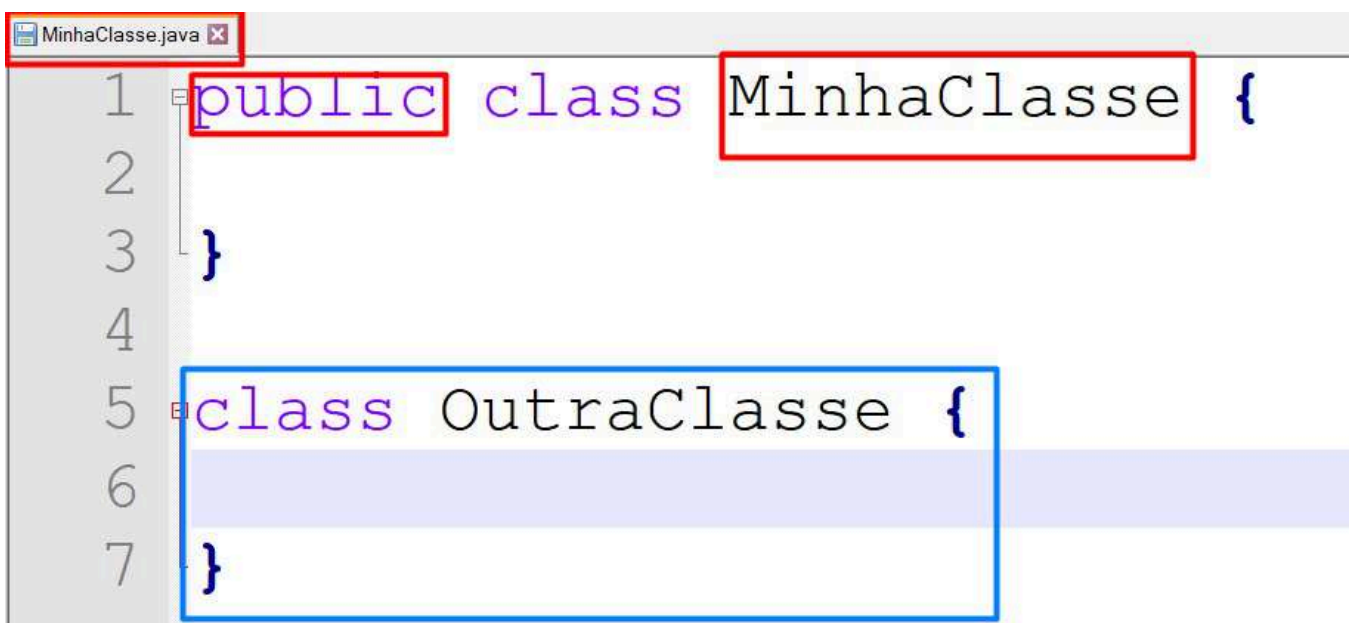
✦ Para fixar

Não é somente o que escreve, mas também como se escreve! Este conteúdo é um complemento das regras de sintaxe apresentadas anteriormente.

2.1.7.1 Classes

Uma classe em Java deve seguir o seguinte padrão:

- O nome da classe *pública* deve igual ao nome do **arquivo.java**;
- Toda classe deve começar com letra maiúscula e se for uma palavra composta, A letra inicial de cada palavra, deverá também ser maiúscula, exemplo: **MinhaClasse**;
- Um arquivo.java poderá ter várias classes, mas somente a classe de mesmo nome, deverá ser `public`. Não recomendamos ter mais de uma classe em um arquivo.java.



```
1 public class MinhaClasse {
2
3 }
4
5 class OutraClasse {
6
7 }
```

2.1.7.2 Atributos

O nome de uma variável, se formado por uma única palavra, deve ser declarado inteiramente em minúsculo. No caso de palavras compostas, apenas a primeira letra de

cada palavra, a partir da segunda palavra, deve estar em maiúsculo. Não é muito recomendado, mas se necessário, usar números ou caracteres especiais em uma variável você deve se estender a estas regras:

- Uma variável não pode iniciar com número;
- Uma variável só pode conter os caracteres: "_" (underline) e "\$" (cifrão);
- Uma variável não ter o nome de uma palavra reservada;
- Uma variável não pode conter espaços;
- Uma variável que representa um conjunto de valores, como array ou coleção, deve estar no formato de plural;
- Todavia, variáveis cujo nome tenha apenas um caractere, deve ser evitado, com exceção de variáveis temporárias, utilizadas em laços de repetição, onde geralmente usamos i, j ou k.

Variáveis válidas

```
1  int um = 1;
2  int numeroUm = 1;
3  int _numeroUm = 1;
4  int $um = 1;
5  int [] numeros = {1,2,3} // ver conceito de arrays
```

java

Variáveis inválidas

```
1  int 1 = 1;
2  int 1um = 1;
3  int %um = 1;
4  int numero um = 1;
```

java

2.1.7.3 Constantes

Nós já aprendemos que existem dois tipos de variáveis, as que podemos alterar o seu valor e as que seu valor não pode ser modificado que denominamos de constante.

Um exemplo comum, é determinar a média mínima de aprovação de alunos de uma escola, como valor 7. A nota dos alunos serão modificadas, mas o valor da média mínima não. Tornando esta variável `final`, nós conseguimos garantir que seu valor não será

modificado, e para explicitar esta regra, nós utilizamos o padrão onde todo o nome da variável será composta por letra maiúscula.

java

```
1 public class NotaFinal {
2     public static void main(String[] args) {
3         int notaUm      = 8;
4         int notaDois     = 7;
5         int notaTres     = 6;
6         int notaQuatro   = 9;
7
8         int mediaFinal   = (notaUm + notaDois + notaTres + notaQuatro) / 4;
9
10        final int NOTA_MINIMA = 7;
11
12        if(mediaFinal >=NOTA_MINIMA)
13            System.out.println("Aprovado");
14        else
15            System.out.println("Reprovado");
16
17    }
18 }
```

Atenção

O que determina se uma variável não pode ser alterada é a palavra reservada **final**, o fato da variável ser totalmente maiúscula é uma adoção das boas práticas de escrita em Java.

2.1.7.4 Métodos

Os métodos, deverão ser nomeados com verbos, através de uma mistura de letras minúsculas e maiúsculas. Em princípio, todas as letras que compõem o nome devem ser mantidas em minúsculo, com exceção, a primeira letra, da segunda palavra composta.

Exemplos sugeridos para nomenclatura de métodos:

java

```
1 int somar(int n1, int n2){ return ... } // métodos podem retornar alguma valor
2
3 abrirConexao() throws Exception{} // este método diz que ao ser executado poderá
4
5
```

```
5 void concluirProcessamento() {} //nem sempre se alguma resposta, logo o retorno
6
7 findById(int id){} // não se assuste, você verá muito método em inglês em sua jo
8
9 calcularImprimir(){} // há algo de errado neste método, ele deveria ter uma únic
10
```

2.1.7.5 Getters e Setters

É comum nossos objetos terem atributos ou características que podem ser definidos ou obtidos os seus valores/estado através da instância de um objeto.

Vamos ver o código abaixo, da criação de um objeto Aluno, com nome e idade:

```
1 //arquivo Aluno.java
2 public class Aluno {
3     String nome;
4     int idade;
5 }
6
7 //arquivo Escola.java
8 public class Escola {
9     public static void main(String[] args) {
10         Aluno felipe = new Aluno();
11         felipe.nome="Felipe";
12         felipe.idade = 8;
13
14         System.out.println("O aluno " + felipe.nome + " tem " + felipe.idade + "
15         //RESULTADO NO CONSOLE
16         //O aluno Felipe tem 8 anos
17     }
18 }
```

Seguindo a convenção Java Beans, uma classe que contém esta estrutura de estados, deverá seguir as regras abaixo:

- Os atributos precisam ter o modificador de acesso `private` . Ex.: `private String nome;`

- Como agora os atributos estarão somente a nível de classe, precisaremos dos métodos **getX** e **setX**, Ex.: `getNome()` e `setNome(String novoNome)`;
- O método **get**, é responsável por obter o valor atual do atributo, logo, ele precisa ser `public` e retornar um tipo correspondente ao valor, Ex.: `public String getNome() {}` ;
- O método **set**, é responsável por definir ou modificar o valor de um atributo, em um objeto, logo, ele também precisa ser `public` , receber um parâmetro do mesmo tipo da variável, mas não retorna nenhum valor `void`. Ex.: `public void setNome(String novoNome) ;`

java

```
1 //arquivo Aluno.java
2 public class Aluno {
3     private String nome;
4     private int idade;
5
6     public String getNome() {
7         return nome;
8     }
9     public void setNome(String novoNome) {
10         nome = novoNome;
11     }
12     public int getIdade() {
13         return idade;
14     }
15     public void setIdade(int newIdade) {
16         this.idade = newIdade;
17     }
18 }
19 //arquivo Escola.java
20 public class Escola {
21     public static void main(String[] args) {
22         Aluno felipe = new Aluno();
23         felipe.setNome("Felipe");
24         felipe.setIdade(8);
25
26         System.out.println("O aluno " + felipe.getNome() + " tem " + felipe.getI
27     }
28 }
```

A proposta do código acima é a mesma que o código anterior, a diferença é que adotamos a convenção Java Beans, para definir e obter as características dos nossos objetos.

Uso do `this` no método `set`, é muito comum vermos nossos métodos de definição, ter a seguinte sintaxe:

```
1 //arquivo Aluno.java
2 private String nome;
3
4 public void setNome(String nome) {
5     this.nome = nome;
6 }
```

java

Atenção

Observe que, a descrição do nosso atributo `nome` é igual a descrição do parâmetro, logo, utilizamos mais uma palavra reservada `this` para distinguir um do outro.

Projeto Lombok

O Lombok é uma biblioteca Java focada em produtividade e redução de código boilerplate que, por meio de anotações adicionadas ao nosso código, ensinamos o compilador (maven ou gradle) durante o processo de compilação a criar código Java.



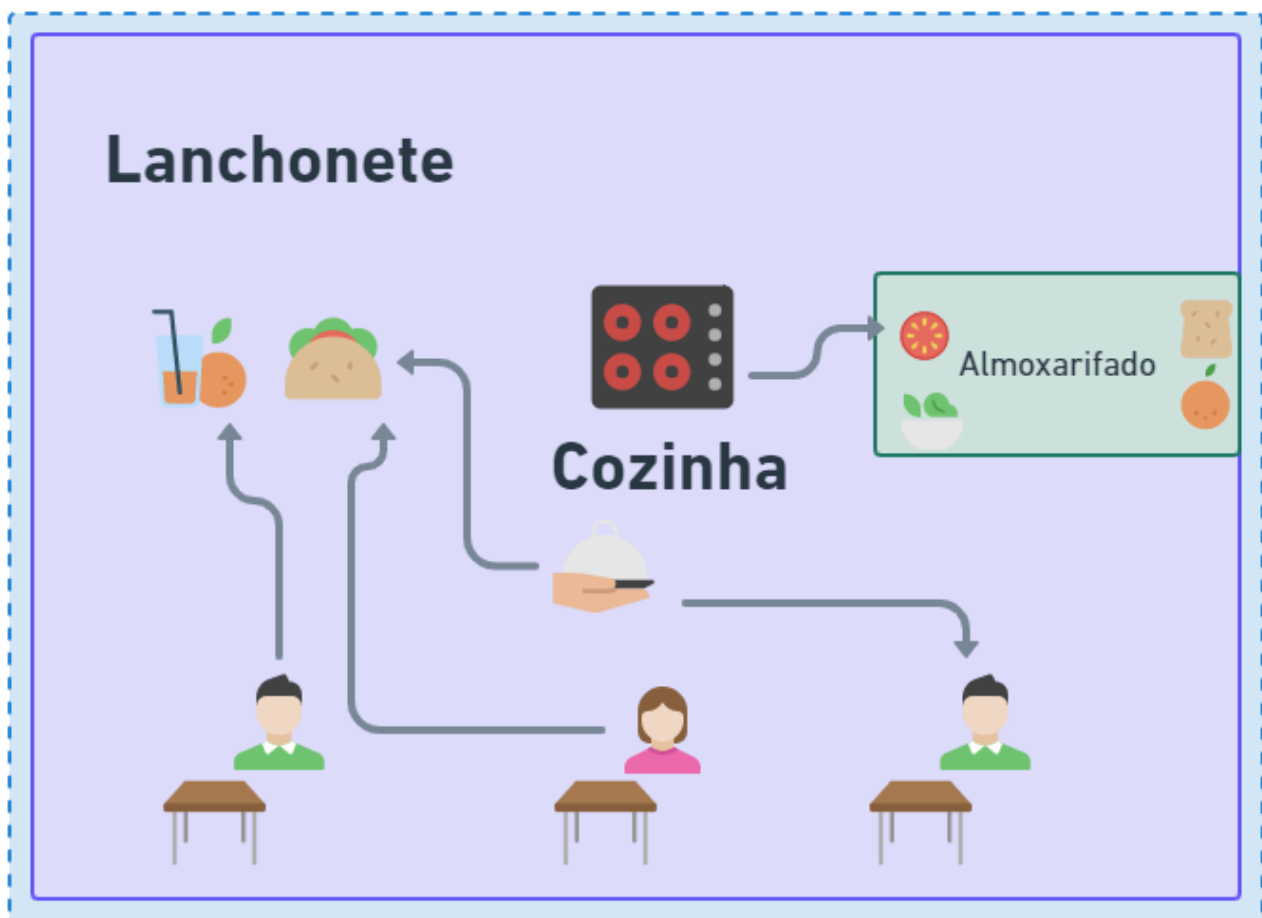
2.1.8 - Modificadores

Em Java, utilizamos três palavras reservadas e um conceito default (sem nenhuma palavra reservada) para definir os quatro tipos de visibilidade de atributos, métodos e até mesmo classes, no que se refere ao acesso por outras classes. Iremos ilustrar do mais visível ao mais restrito tipo de visibilidade nos arquivos em nosso projeto.

Para uma melhor ilustração iremos representar os conceitos de visibilidade de recursos através do contexto em uma lanchonete que vende lanche natural e suco.

2.1.8.1 Modificador public

Como o próprio nome representa, quando nossa classe, método e atributo é definido como public, qualquer outra classe em qualquer outro pacote, poderá visualizar tais recursos.



Primeiro, iremos criar o pacote **lanchonete**

Estabelecimento.java

Cozinheiro.java

Almoxarife.java

Atendente.java

Cliente.java

java

```
1 package lanchonete;
2
3 public class Estabelecimento {
4     public static void main(String[] args) {
5         Cozinheiro cozinheiro = new Cozinheiro();
6         //ações que não precisam estarem disponíveis para toda a aplicação
7         cozinheiro.lavarIngredientes();
8         cozinheiro.baterVitaminaLiquidificador();
9         cozinheiro.selecionarIngredientesVitamina();
10        cozinheiro.prepararLanche();
11        cozinheiro.prepararVitamina();
12        cozinheiro.prepararVitamina();
13
14        //ações que estabelecimento precisa ter ciência
15        cozinheiro.adicionarSucoNoBalcao();;
16        cozinheiro.adicionarLancheNoBalcao();
17        cozinheiro.adicionarComboNoBalcao();
18
19        Almoxarife almoxarife = new Almoxarife();
20        //ações que não precisam estarem disponíveis para toda a aplicação
21        almoxarife.controlarEntrada();
22        almoxarife.controlarSaida();
23        //ação que somente o seu pacote cozinha precisa conhecer (default)
24        almoxarife.entregarIngredientes();
25        almoxarife.trocarGas();
26
27        Atendente atendente = new Atendente();
28        atendente.pegarPedidoBalcao();
29        atendente.receberPagamento();
30        atendente.servindoMesa();
31
32        Cliente cliente = new Cliente();
33        cliente.escolherLanche();
34        cliente.fazerPedido();
35        cliente.pagarConta();
36
37        //não deveria, mas o estabelecimento
38        //ainda não definiu normas de atendimento
39    }
```

```
40     cliente.pegarPedidoBalcao();
41
42     //esta ação é muito sigilosa, qua tal ser privada ?
43     cliente.consultarSaldoAplicativo();
44
45     //já pensou os clientes ouvindo que o gás acabou ?
46     cozinheiro.pedirParaTrocarGas(almoxarife);
47
48 }
```

📌 Para fixar

Acredite! Nem tudo precisa ser visto por todos. 😊

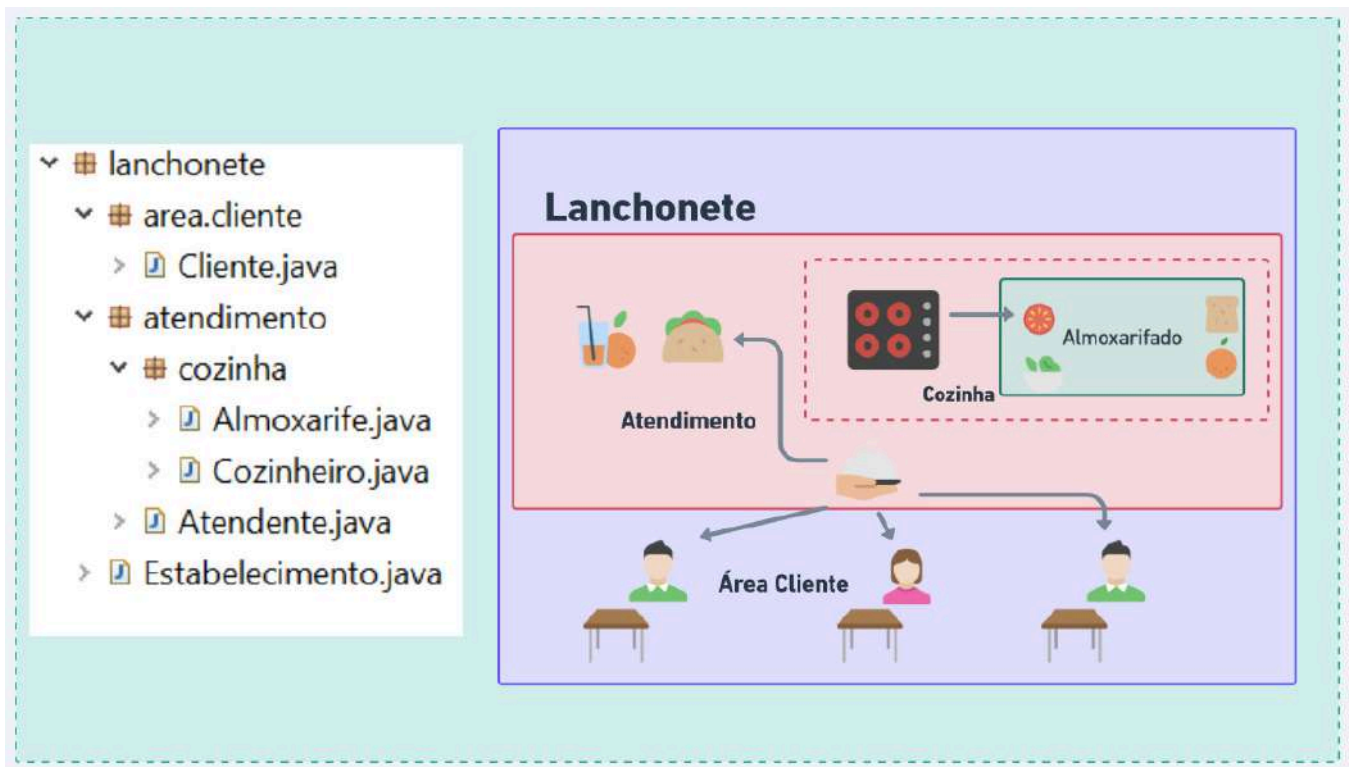
2.1.8.2 Modificador default

O modificador `default`, está fortemente associado a organização das classes por pacotes ONDE algumas implementações não precisam estar disponíveis para todo o projeto, e este modificador de acesso, restringe a visibilidade por pacotes.

Dentro do pacote `lanchonete`, iremos criar dois sub-pacotes para representar a divisão do estabelecimento.

- **lanchonete.atendimento.cozinha:** Pacote que contém classes da parte da cozinha da lanchonete e atendimentos.
- **lanchonete.area.cliente:** Pacote que contém classes relacionadas ao espaço do cliente.

Após criar os pacotes, em sua IDE mova as as classes conforme imagem ilustrativa abaixo:



Hora de praticar a visibilidade de nossos recursos

```
1 // Cozinheiro.java
2
3 //ANTES
4 public void lavarIngredientes() {
5     System.out.println("LAVANDO INGREDIENTES");
6 }
7
8 //nível de pacote
9 //sem nenhuma palavra reservada de acesso
10 void lavarIngredientes() {
11     System.out.println("LAVANDO INGREDIENTES");
12 }
```

java

🔔 Atenção

Alguns erros poderão ser apresentados na sua tela, não se preocupe iremos corrigi-los.

2.1.8.3 Modificador private

Depois de reestruturar nosso estabelecimento (projeto), onde temos as divisões (pacotes), espaço do cliente e atendimento, chegou a hora de uma reflexão sobre

visibilidade ou modificadores de acesso.

Conhecemos as ações disponíveis nas classes `Cozinheiro`, `Almoxarife`, `Atendente` e `Cliente`, mesmo com a organização da visibilidade por pacote, será que realmente estas classes precisam ser tão explícitas?

- Será que o `Cozinheiro` precisa saber como o `Almoxarife` controla as entradas e saídas ?
- Que o `Cliente` precisa saber como o `Atendente` recebe o pedido para servir sua mesa ?
- Que o `Atendente` precisa saber que antes de pagar o `Cliente` consulta o saldo no App ?

Diante destes questionamentos, é que nossas classes precisam continuar mantendo suas ações (métodos), mas nem todas precisam ser vistas por alguém.

✓ Conclusão

A visibilidade de recursos da linguagem não está associada a **interface gráfica**, mas sim, o que as classes conseguem **acessar** umas das outras. 🤔

2.1.8.4 Modificador `protected`

O que precisamos compreender sobre o modificador `protected` inicialmente? Que o mesmo está fortemente relacionado a herança, um dos princípios disponível no conceito de **P O O - Paradigma da Orientação à Objetos**.