

Material Workshop - Fundamentos do Java

Antes de adentrarmos nos tópicos em si, é válido ressaltar a estrutura básica de quando iniciamos um novo projeto em Java.

```
Java
package exemploA;

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

    }
}
```

Sempre que você iniciar um novo projeto em Java, essa será a estrutura padrão. A palavra `package` indica dentro de qual pasta nós estamos dentro do nosso projeto. Imagine-se dentro do seu computador na área de trabalho, você deve ter várias pastas, sendo que dentro de cada pasta você pode ter mais pastas ou arquivos. Dentro da programação, nós fazemos esse mesmo tipo de organização. O número de pastas que sua solução vai conter depende do tamanho do problema que vai ser resolvido e qual a arquitetura utilizada. Porém, não se preocupe com isso agora. Tenha em mente que podemos nomear de qualquer forma, neste exemplo, eu nomeei de “exemploA”.

Um pouco mais abaixo, temos a definição da nossa classe. Uma vez que o `package` representa a nossa pasta, as classes, a grosso modo, irão representar os arquivos. Da mesma forma que os `packages`, nós podemos nomear as classes da forma que for mais conveniente, porém, como boa prática, normalmente a primeira letra é sempre maiúscula, como pode-se observar no exemplo acima “ClasseA”.

Dentro do nosso arquivo `ClasseA`, temos um trecho de código que parece meio estranho. Não se acanhe em entender todos os significados de `public`, `static`, `void` e `args` por agora. Neste momento, o que você precisa saber desse método é que ele é como a nossa mãe. Faça uma reflexão consigo mesmo, dentro de sua casa, quem manda? Se você respondeu seu pai, você está enganado. É a sua mãe! Logo, em

analogia à palavra “main”, essa função é a mãe do nosso código, ou seja, tudo começa a ser executado a partir dela.

Variáveis e Tipos de Dados

Agora que já vimos a estrutura básica, vamos entrar nos tópicos mais específicos da linguagem. Começando pelo conceito de variável, irei pedir para que você coloque sua imaginação em jogo e permita-se simular o seguinte cenário: eu irei falar dois números para você e quero que você guarde bem eles, vamos lá? Primeiro, eu quero que você guarde o número a que recebe o valor 1 e, por fim, eu quero que você guarde, também, o número b que recebe 2. Neste momento, eu quero que você realize a soma desses 2 números, o que deve resultar em 3, correto? Mas como você conseguiu fazer essa soma na sua cabeça? Como você lembrou dos números enquanto realizava a operação?

Bom, o que aconteceu é que, quando eu falei os números, você reservou um espaço na sua memória para cada número e, dentro desse espaço, você colocou um valor. Basicamente, foram criados dois espaços (espaço a e espaço b) que receberam, respectivamente, 1 e 2. Assim, uma variável nada mais é do que um espaço na memória do seu computador que recebe um nome e um valor.

Em Java, nós podemos criar variáveis da seguinte forma:

```
Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        int a = 1;
        int b = 2;
    }
}
```

Em analogia aos números a e b que você guardou na sua memória, nós representamos isso dentro do nosso código agora. Um ponto importante é que eu pedi para você guardar um número inteiro, mas eu poderia ter pedido outro tipo, como um texto ou um número quebrado também. Na sua cabeça, pouco importa se

o número é inteiro ou quebrado para guardá-lo, porém, dentro da programação, isso importa! Portanto, quando nós vamos criar uma variável dentro do Java, nós precisamos definir o que chamamos de tipo de dado.

O tipo de dado representa qual é o tipo da informação que está sendo guardada naquele espaço de memória. Na imagem acima, o tipo de dado é o int, o qual representa uma parcela dos números inteiros dentro do java. Mas, não paramos por aqui, é possível representar outros tipos de dados também:

```
Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        // int é o tipo de dado que representa um número inteiro
        int a = 1;

        // double é o tipo de dado que representa um número quebrado
        double b = 2.2;

        // String é o "tipo de dado" que representa um texto
        String nome = "henrique";

        // char é o tipo de dado que representa um caracter
        char c = 'h';

        // boolean é o tipo de dado que representa verdadeiro/falso
        boolean palmeirasTemMundial = false;
    }
}
```

Acima, temos algumas formas de representar os tipos de dados dentro do Java. Observe que, em cima da criação das nossas variáveis, foi colocado // seguido de um texto. Isso é o que chamamos de comentário. Um comentário é como uma cápsula do tempo para si mesmo, ou então, para alguém que irá trabalhar na sua equipe de desenvolvimento. Os comentários são utilizados para explicar trechos de código, o que irá facilitar uma futura revisão daquela funcionalidade comentada.

Você pode fazer um comentário de linha única, um comentário de várias linhas ou um comentário voltado para a documentação. Na imagem abaixo, você pode verificar a instrução de cada um.

```

Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        // comentário de linha única

        /*
            comentário
            de
            várias
            linhas
        */

        /**
            comentário
            voltado para
            documentação
            (você pode adicionar tags aqui)
        */
    }
}

```

Dentro do nosso código, os comentários são ignorados, ou seja, eles não interferem no fluxo do seu código (ao menos que você comente um bloco de código).

Operadores

No exemplo que utilizamos alguns blocos acima, eu pedi para que você guardasse os dois números e realizasse a soma deles, correto? Será que conseguimos implementar essa operação de soma dentro do nosso código? A resposta é sim. Dentro das linguagens de programação (e com o Java não seria diferente), nós temos os operadores aritméticos. Esses operadores são utilizados para representar cálculos matemáticos dentro do nosso programa.

```

Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        int a = 1;
    }
}

```

```

        int b = 2;
        int soma = a + b;
    }
}

```

Dentro do bloco de código acima, temos a criação de uma nova variável que está armazenando o resultado da operação de soma entre nossas variáveis a e b. Vamos fazer esse resultado aparecer na nossa tela agora?

```

Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        int a = 1;
        int b = 2;
        int soma = a + b;

        System.out.println(soma);
    }
}

```

```

Unset
3

```

Ao que tudo indica, nosso programa está funcionando corretamente. Porém, as operações aritméticas são feitas apenas através de variáveis? Vamos testar!

```

Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        int a = 1;
        int b = 2;
        int soma = a + b;

        System.out.println(a + b);
    }
}

```

```
}
```

Agora, ao invés de colocarmos a variável soma dentro da nossa impressão, nós estamos passando diretamente a soma entre as variáveis a e b.

```
Unset  
3
```

Ao rodar o código, temos a mesma saída. Ou seja, podemos realizar as operações diretamente através de números e até mesmo em textos. Vamos tentar “somar” dois textos agora?

Primeiro, vamos começar definindo duas novas variáveis:

```
Java  
package exemploA;  
  
public class ClasseA {  
    public static void main (String[] args) {  
        String a = "Escola";  
        String b = "Evolua";  
    }  
}
```

Uma vez que as variáveis estão definidas, vamos tentar imprimir a “soma delas”:

```
Java  
package exemploA;  
  
public class ClasseA {  
    public static void main (String[] args) {  
        String a = "Escola";  
        String b = "Evolua";  
  
        System.out.println(a + b);  
    }  
}
```

O resultado da saída é:

```
Unset  
EscolaEvolua
```

Se observar bem, você pode ver que eu deixei a palavra soma entre aspas. Isso se deve por conta que, dentro de Strings, essa operação é denominada de concatenação. Portanto, sempre que juntamos alguma variável, independente do tipo de dado, com uma String, a operação é chamada de concatenação.

No exemplo acima, o nosso texto “EscolaEvolua” saiu colado, vamos adicionar um caractere de espaço entre as Strings?

```
Java  
package exemploA;  
  
public class ClasseA {  
    public static void main (String[] args) {  
        String a = "Escola";  
        String b = "Evolua";  
  
        System.out.println(a + " " + b);  
    }  
}
```

```
Unset  
Escola Evolua
```

O que aconteceu foi o seguinte: foram realizadas duas concatenações. Primeiro, a variável a foi concatenada com o espaço e, esse valor resultante da concatenação, foi concatenado com a variável b.

Até agora você só viu o operador de soma, porém, não paramos por aqui. Nós podemos realizar operações de subtração, multiplicação, divisão e resto também:

```
Java  
package exemploA;  
  
public class ClasseA {
```

```
public static void main (String[] args) {  
    int a = 1;  
    int b = 2;  
    int soma = a + b;  
    int subtracao = a - b;  
    int multiplicacao = a * b;  
    double divisao = a / b;  
    int resto = a % b;  
}  
}
```

Todos os operadores estão presentes no dia a dia e, por conta disso, não irei me aprofundar. O que temos de diferente é o operador de resto. Ao fazer uma divisão, ela pode ser exata ou não exata. Quando ela é exata, temos um resto zero, quando ela não é exata, o resto pode variar. Por exemplo, ao dividir 3 por 2, tem o resto 1. Neste meio, o operador % indica essa operação de resto dentro do Java.

Estrutura Condicional

Tudo na nossa vida se baseia na tomada de decisões e dentro do nosso código não seria diferente. Imagine-se no seguinte cenário: você sai cedo de casa para ir até o mercado comprar seu pão para tomar café, porém, no meio deste caminho, você tem que atravessar uma avenida. Primeiro, você verifica se está vindo carro de um lado, se não estiver, agora você verifica se está vindo algo do outro lado também. Se nenhum carro estiver vindo em ambas as direções, você atravessa a rua.

Qual foi o processo para a sua tomada de decisão? Você analisou as duas mãos da avenida, viu que não haviam carros vindo em nenhuma delas e então tomou a sua decisão de atravessar a rua.

A estrutura sempre será essa, você tem o que está sendo verificado (está vindo carro?) e a ação a ser tomada dependendo da verificação final (se não estiver, atravesse a rua).

Dentro do Java, essa estrutura de condição é representada pelo bloco if. Vamos começar com a estrutura básica desse bloco verificando se dois números são iguais:


```

Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        int a = 2;
        int b = 2;

        if (a == b) {
            System.out.println("O número a é igual ao número b!");
        }
    }
}

```

Passando o que está escrito em Java para uma linguagem falada, podemos chegar à seguinte frase: “se a for igual a b, então imprima na tela a mensagem ‘O número a é igual ao número b!’ ”.

Basicamente, nós estamos verificando o que está dentro dos parênteses depois do if. No exemplo acima, estamos verificando se o valor de a é igual ao valor de b. Você deve estar estranhando o uso dos dois símbolos de =, certo? Mas, fique tranquilo, não há nenhum erro ali. Ele é operador relacional de igualdade. Acima, eu comentei apenas sobre os operadores aritméticos, mas, nós temos também os operadores relacionais, lógicos e de atribuição. Neste momento, não há grande necessidade de entrar em muitos detalhes, entretanto, os operadores relacionais são utilizados para testar a relação entre duas entidades. Abaixo, você pode verificar na tabela quais operadores relacionais existem no Java.

Operador	Comparação
==	Igual
!=	Diferente
<	Menor
<=	Menor igual
>	Maior
>=	Maior igual

No exemplo acima, estamos tomando uma ação apenas se a condição entre parênteses for verdadeira, porém, e se também quisermos tomar uma ação caso a condição for falsa? Para isso, temos o else. Em momentos anteriores, eu chamei de bloco if apenas para descomplicar, mas você sempre irá ouvir por bloco if-else. Agora, vamos colocar uma mensagem para indicar que o número a é diferente do número b.

```
Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        int a = 2;
        int b = 2;

        if (a == b) {
            System.out.println("O número a é igual ao número b!");
        } else {
            System.out.println("O número a é diferente do número
b!");
        }
    }
}
```

Bem melhor! Se em algum momento o nosso valor de a ou de b mudar, podemos verificar se os números são diferentes. Isso foi possível por conta da adição do else. Para mudar um pouco, vamos fazer um exemplo para verificar se um número é par ou ímpar.

```
Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        int a = 2;

        if (a % 2 == 0) {
            System.out.println("O número é par!");
        } else {
            System.out.println("O número é ímpar!");
        }
    }
}
```

Vamos tentar entender o código acima. Começamos com o seguinte questionamento: quando um número é par e quando um número é ímpar? De prontidão e de forma simples, podemos indicar um número como par quando a divisão dele por dois resulta em uma operação de resto 0. E, para um número ser ímpar, ocorre o processo contrário, ou seja, a divisão do número por 2 resulta em um resto maior que zero. Por conta disso, utilizamos o operador de resto na nossa condição, onde estamos verificando se o resto da divisão entre a nossa variável `a` por 2 é igual a 0. Se for, a mensagem impressa na tela será: “O número é par!”; caso contrário, irá aparecer: “O número é ímpar!”.

Rodando o código, com `a` valendo 2, tem-se o seguinte resultado:

```
Unset  
0 número é par!
```

Agora, vamos alterar o valor de `a` para 3 e rodar novamente:

```
Unset  
0 número é ímpar!
```

O nosso resultado mudou. Essa mudança representa a tomada de decisão dentro do nosso código!

Estrutura de Repetição

Dentro da programação, haverá momentos em que você deseja repetir algumas vezes um determinado trecho de código e a pergunta que fica é: como você iria fazer isso? Caso o número de repetições seja 5, você vai copiar e colar 5 vezes aquele código?

É nesse contexto que entram as estruturas de repetição. Antes de entrarmos em detalhes mais técnicos, vamos - novamente - utilizar um paralelo com o mundo real. Suponha que você tenha uma gaveta cheia de lápis coloridos dentro do seu quarto

e eu peço para que você ache o lápis amarelo para mim. Como funcionará esse algoritmo?

Ao abrir a gaveta, você vai pegar lápis por lápis e analisar a cor dele, caso a cor do lápis seja amarela, você para. Caso contrário, você continua. Basicamente, o comando é o seguinte: enquanto a cor do lápis não for amarela, continue.

Uma estrutura de repetição é isso. Ela irá testar uma condição e, enquanto essa condição for verdadeira, o bloco de código dentro da estrutura será repetido. O número de repetições pode ser indefinido ou definido. Para cada um desses casos, iremos utilizar uma estrutura diferente.

Vamos começar pelo seguinte exemplo: suponha que dentro de uma gaveta você tenha 30 lápis e você queira tirar 20. Como isso poderia ser feito?

```
Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        // criação da variável responsável por guardar a quantidade de
        lápis
        int contagemLapis = 30;

        // laço de repetição para tirar 20 lápis
        for (int i = 0; i < 20; i++) {
            contagemLapis = contagemLapis - 1;
        }

        // imprimindo o resultado que será 10
        System.out.println(contagemLapis);
    }
}
```

Para resolver esse exemplo, foi utilizado o laço de repetição for. O for possui 3 “elementos” dentro dele. A primeira parte, `int i = 0`, é a criação da nossa variável de controle dentro do laço, ela será a responsável por armazenar quantas vezes já repetimos o laço. A segunda parte, `i < 20`, é a nossa condição de parada, ou seja, é a condição que tem que ser verdadeira para que o bloco dentro do for seja executado. E, por último, temos a parte de incremento (`i++`)/decremento (`i—`) da

nossa variável de controle. Sempre que o bloco de código foi executado, o for volta para cima, atualiza o valor da variável de controle e passa para a verificação da condição.

Além do for, temos também o laço de repetição while. O while é composto apenas de uma condição e o bloco de código que deve ser executado. Vamos resolver o mesmo problema proposto acima, só que agora com o while:

```
Java
package exemploA;

public class ClasseA {
    public static void main (String[] args) {
        // criação da variável responsável por guardar a quantidade de
        lápis

        int contagemLapis = 30;

        // laço de repetição para tirar 20 lápis
        while (contagemLapis > 10) {
            contagemLapis = contagemLapis - 1;
        }

        // imprimindo o resultado que será 10
        System.out.println(contagemLapis);
    }
}
```

O código ficou mais simples, não é mesmo? Porém, nem sempre iremos querer isso. Há problemas que serão resolvidos utilizando o for e há problemas que serão resolvidos utilizando o while. A sacada virá com o tempo e com o acúmulo de problemas resolvidos. Mas, grosseiramente falando, a diferença é que utilizamos o while quando não sabemos quantas repetições serão necessárias, já o for será utilizado para quando este número for conhecido.

Métodos

Em analogia ao mundo real, podemos encarar um método como uma função matemática. Basicamente, uma função recebe valores, faz algum tipo de cálculo e nos retorna um resultado. Dentro da programação, os métodos podem seguir literalmente todas essas 3 etapas do processo, ou então, apenas algumas partes. É

fato que todos os métodos devem conter, no mínimo, a etapa de processamento. Essa etapa condiz com o funcionamento do seu método, ou seja, ali estará toda a “inteligência” e lógica. Nossos métodos podem conter valores de entrada, que são chamados de parâmetros e podem fornecer algum tipo de saída através de um retorno. Quando não temos esse tipo de retorno, declaramos o tipo da nossa função como void. Mas, o que seria esse tipo da função? Da mesma forma que declaramos os tipos de nossas variáveis, é necessário declarar o tipo do nosso método. Caso o nosso método retorne um texto, nós o declaramos como String; se o método retornar um número inteiro, declaramos como inteiro e aí por diante.

Para criar uma função, iremos seguir a seguinte sintaxe:

```
Java
public int somarValores(int a, int b) {
    int soma = a + b;
    return soma;
}
```

Vamos começar dando nomes às partes pertencentes da primeira linha acima, a qual chamamos de cabeçalho da função (leia-se função como outro nome para método):

`public int somarValores(int a, int b)`

Como indicado na figura, o cabeçalho é dividido em 4 partes:

- **Visibilidade do método:** a visibilidade indica em quais parte do nosso código a função será visível. Abaixo, estão contidos os tipos de visibilidades e sua atuação.

Tipo de visibilidade	Para quem é visível
Public	Todas as classes que contém esta classe em uso
Protected	Classe mãe (superclasse) e classes filhas (subclasses)
Private	Classe mãe (superclasse)

- **Tipo de retorno da função:** o tipo de retorno da nossa função é análogo ao tipo de variável. Da mesma forma que uma variável int não pode conter um número double, um método declarado como int não pode retornar um número que for double. Os métodos irão seguir os mesmos tipos de dados que mencionamos posteriormente dentro de variáveis.
- **Nome do método:** ao criar um método, precisamos definir um nome para ele. Isso ocorre, pois, quando queremos executar o que está dentro do nosso método, nós precisamos chamá-lo. A forma que um método é chamado é através de seu nome seguido de (). Dentro desses parênteses, deve-se colocar os parâmetros que a função deseja, ou, caso você não tenha parâmetros, deixe em branco.

Abaixo, confira um exemplo da chamada da função que criamos acima:

Java

```
somarValores(3, 4);
```

Se o nosso método não tivesse parâmetros, a sua chamada seria da seguinte forma:

Java

```
somarValores();
```

- **Parâmetros da função:** os parâmetros indicam tudo o que o nosso método vai precisar para o seu processamento. Lembre-se de qualquer função da matemática ou de física, nós não conseguimos resolver elas sem que tenhamos algumas variáveis específicas. Essas variáveis dentro da matemática e da física são equivalentes aos nossos parâmetros dentro da programação.

Agora que passamos pela definição do nosso método, nós podemos entrar na parte do processamento em si. Basicamente, o processamento é o que contém toda a lógica da nossa função. No exemplo, nós fizemos uma função para somar dois

números. O processamento deste exemplo seria a criação da variável soma e a sua atribuição à operação $a + b$, o qual está indicado em vermelho na figura abaixo.

```
public int somarValores(int a, int b) {  
    int soma = a + b;  
    return soma;  
}
```

Você deve estar se perguntando o por que do trecho `return soma` não estar fazendo parte do processamento também. Bem, ele não deixa de estar, porém, ele está mais intimamente ligado ao tipo de retorno do nosso método do que com o processamento em si. O seu método pode não conter esse trecho de retorno, no entanto, quando isso acontecer, lembre-se de declarar o tipo da sua função como void.

Os métodos, ou funções, são muito importantes para o reaproveitamento de código. Imagine uma situação em que certa funcionalidade deve se repetir em 3 lugares distintos do seu código, o que você faria? Você copiaria e colaria todas as linhas de código as 3 vezes, ou então você criaria um método com essas instruções e depois apenas chamaria o método nos 3 lugares desejados? Bom, suponha que essa funcionalidade contenha 100 linhas de código, você teria 300 linhas fazendo a mesma coisa. Desnecessário, certo?

Portanto, o que eu quero que você entenda é o seguinte, os métodos irão servir para a criação de funcionalidades específicas do seu código e, com isso, seu código ficará mais organizado e limpo, seguindo as boas práticas da programação.

Classes e Objetos

Quando entramos dentro de classes e objetos, estamos nos referindo à programação orientada a objetos. Esse paradigma da programação refere-se à abstração do mundo real para dentro do nosso código, sendo que a abstração ocorre através dos objetos. O que nós temos no mundo real pode ser mapeado em características e comportamentos, os quais são passados para a programação através de atributos e métodos. As características envolvem as informações contidas no objeto do mundo real. Por exemplo, imagine que queremos representar

um carro dentro da programação, é fato que todo carro possui uma cor, um ano de fabricação, um modelo, uma marca, quantidade de km rodados e por aí vai...

Agora, quando falamos de comportamento, estamos nos referindo às ações que um objeto pode executar. Continuando no exemplo do carro, um carro pode acelerar, frear, mudar a marcha, ligar e desligar; o que acabamos de fazer foi mapear algumas características e comportamentos que um carro possui e agora podemos levar isso para dentro da programação através de atributos e métodos.

Antes de criarmos os atributos e métodos do nosso carro, vamos ver como é a estrutura para iniciar uma classe:

```
Java
public class Carro {

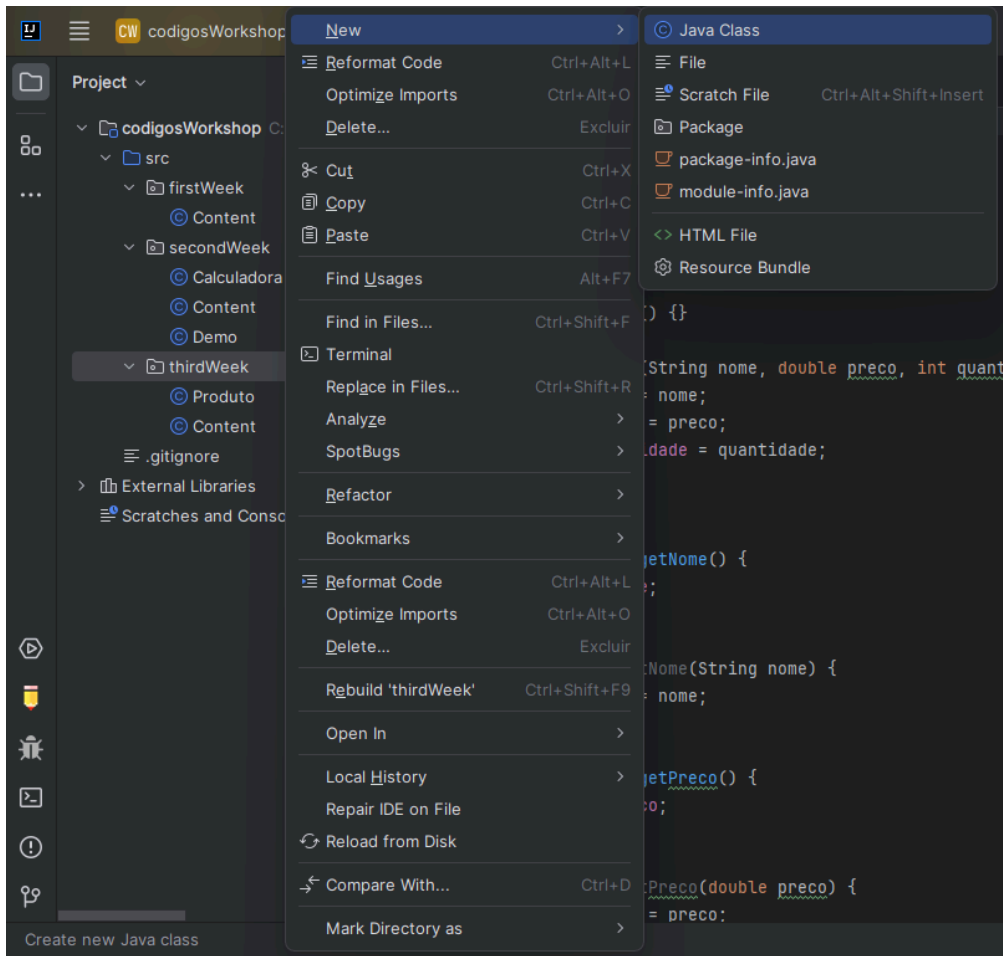
}
```

Essa é a estrutura inicial ao se criar uma nova classe. Note que não temos nenhuma main para executar o nosso código, isso se deve porque nós não executamos nada relacionado à classe, e sim às suas instâncias (objetos).

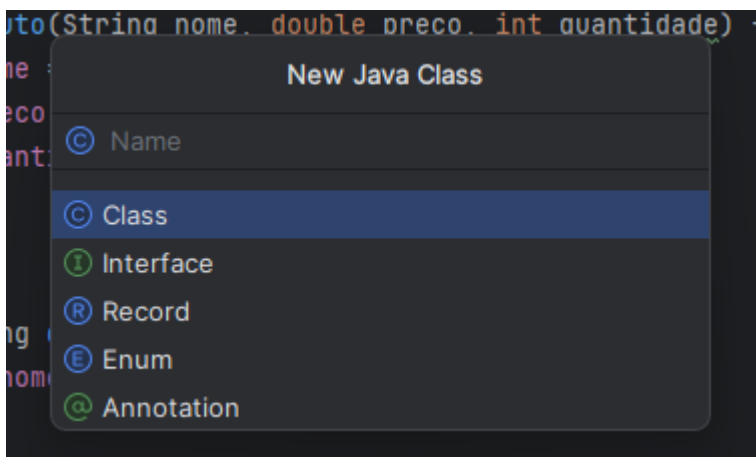
Mas, qual foi o caminho para chegar nesse resultado?

Primeiramente, nós devemos criar uma nova classe na nossa IDE ou em nosso editor de código.

Na imagem abaixo, perceba que dentro do nosso pacote (pasta) thirdWeek, foi pressionado o botão direito do mouse para que a segunda aba se abrisse. Nela, foi selecionada a opção New (Novo) e, por último, a opção Java Class (Classe Java).



Ao clicar na opção indicada acima, a seguinte janela irá se abrir:



Dentro do campo de nome, iremos colocar o nome da nossa classe. Neste caso, colocarei Carro (lembrando que sempre definimos uma classe com a primeira letra maiúscula). Além disso, lembre-se de deixar a opção Class (Classe) selecionada.

Ao pressionar enter, temos a seguinte classe criada:

```
1 package thirdWeek;
2
3 public class Carro {
4     public static void main(String[] args) {
5
6     }
7 }
8 |
```

Não iremos precisar desse método main, então podemos apagá-lo. Ele foi criado por padrão da IDE, neste caso, estou utilizando a IDE IntelliJ IDEA Community.

Para saber mais sobre essa IDE, acesse o link abaixo:

<https://www.jetbrains.com/idea/download/?section=windows>

Após apagar o método main, nossa classe ficará assim:

```
1 package thirdWeek;
2
3 public class Carro {
4
5 }
6
```

Essa é a estrutura inicial para começarmos a trabalhar com a nossa classe. Pode ser que, ao utilizar outra IDE, ela não crie o método main automaticamente, aí basta pular os passos acima.

Uma vez que temos nossa estrutura pronta, precisamos criar nossos atributos e métodos:

- **Atributos:** os atributos da nossa classe serão representados através de variáveis, onde iremos declarar-las da seguinte forma:

`public int nomeAtributo;`

- **Visibilidade do atributo:** a visibilidade de um atributo irá funcionar da mesma forma que a visibilidade de um método. Essa visibilidade será dividida em 3 formas:

Tipo de visibilidade	Para quem é visível
Public	Todas as classes que contém esta classe em uso
Protected	Classe mãe (superclasse) e classes filhas (subclasses)
Private	Classe mãe (superclasse)

- **Tipo de dados do atributo:** como estamos definindo uma variável, é necessário definir qual será o tipo de dados dela, como por exemplo: int, double, String, char, boolean...
- **Nome do atributo:** o nome do atributo refere-se ao nome da variável em si. Mais adiante, mostrarei que o nosso intuito, por conta de alguns dos pilares da orientação a objetos, é não acessar os nossos atributos diretamente através do seu nome.
- **Métodos:** os métodos que representam os comportamentos do nosso objeto seguem os mesmos princípios explanados durante o tópico de métodos.

Agora que você já sabe sobre como abstrair algo do mundo real para dentro do código, vamos ver como isso fica:

```
Java
public class Carro {
    // definindo nossos atributos
    private String cor;
    private String marca;
    private String modelo;
    private int ano;
}
```

Você pode notar que deixamos nossos atributos com a visibilidade *private* e deve estar se perguntando o porquê de se fazer isso. Dentro do paradigma da

programação orientada a objetos, nós temos alguns pilares importantes. Um deles é o encapsulamento. O encapsulamento serve para proteger o nosso código de nós mesmos. Mas, uma vez que nossos atributos estão privados, como eu faço para acessá-los? Isso será feito através de métodos que chamamos de getters e setters. Pense no seguinte exemplo: você vai até um banco para pegar o dinheiro que há em sua conta bancária. Para isso, ou você vai até um balcão com um recepcionista, ou então, você entra na sua conta com seu cartão em um caixa eletrônico. Vamos usar como exemplo o caso do balcão com um recepcionista. Ao querer pegar esse dinheiro, você tem o intermediário - recepcionista - que confirma se você é realmente o dono e qual é a sua conta. Após fazer essa consulta, ele retorna esse valor para você. Imagine agora que você deseja depositar certa quantia na sua conta, o processo será parecido. Não é você que vai fazer todo o processo de colocar o dinheiro na sua conta bancária, isso será feito pelo recepcionista. O ponto é o seguinte, os métodos de acesso getter e setter servem como esse recepcionista, ou seja, ele é um intermediário. Você não vai mexer diretamente no seu dinheiro (variáveis), isso será feito através dos recepcionistas (métodos de acesso). Quando você deseja recuperar uma informação, como por exemplo um nome, idade, peso, marca, modelo, cor... você irá usar o método get. Para atribuir um novo valor ao seu objeto, você irá usar o método set. Vamos ver como fica isso dentro do código (lembrando, sua IDE faz tudo isso para você de forma automática).

```
Java
package thirdWeek;

public class Carro {
    // definindo nossos atributos
    private String cor;
    private String marca;
    private String modelo;
    private int ano;

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }
}
```

```

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }

    public String getCor() {
        return cor;
    }

    public void setCor(String cor) {
        this.cor = cor;
    }

    public int getAno() {
        return ano;
    }

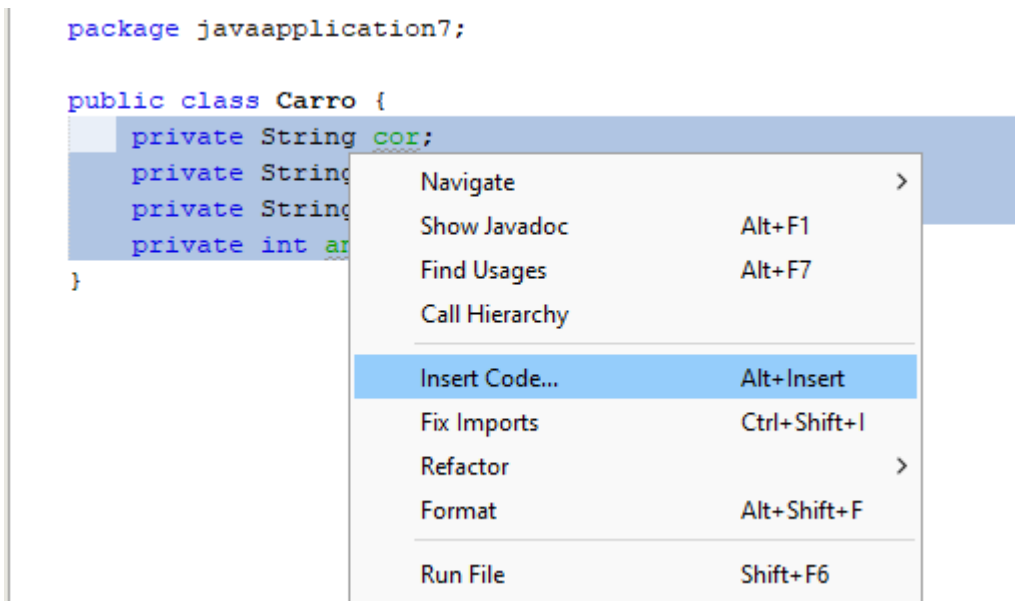
    public void setAno(int ano) {
        this.ano = ano;
    }
}

```

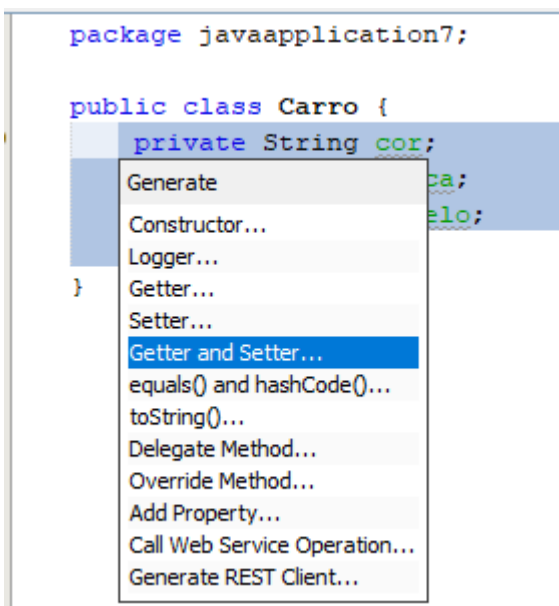
Como você pode observar, nossos atributos estão com a visibilidade `private` e os métodos estão `public`. Todo o acesso às nossas informações será feito através desses métodos públicos. Esse é o conceito de encapsulamento.

Aliás, é uma boa prática deixar todos os métodos da sua classe com a visibilidade `public`, ao menos que seja uma função auxiliar à outra função da classe, neste caso a visibilidade não irá importar muito.

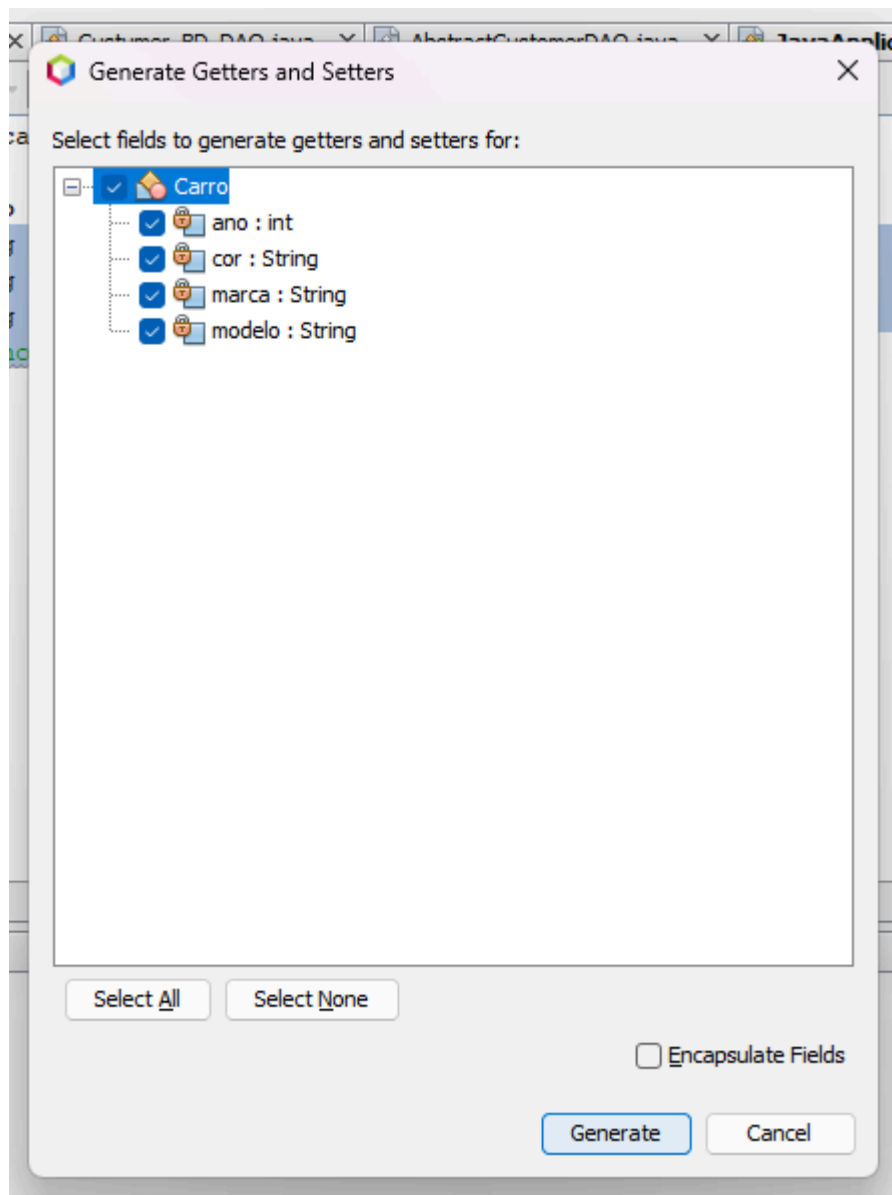
Para criar os métodos de forma automática utilizando a IDE você pode seguir o seguinte passo a passo (estarei utilizando o IntelliJ, porém em outras IDE's a ideia é a mesma). Inclusive, em muitas IDE's podemos criar todos os getters e setters com apenas um clique, bastando apenas selecionar todos os atributos, clicar com o botão direito em cima da seleção e, no caso da IDE NetBeans, selecionar a opção `Insert Code`:



Ao clicar em Insert Code, o seguinte menu se abrirá e escolheremos a opção Getter and Setter:



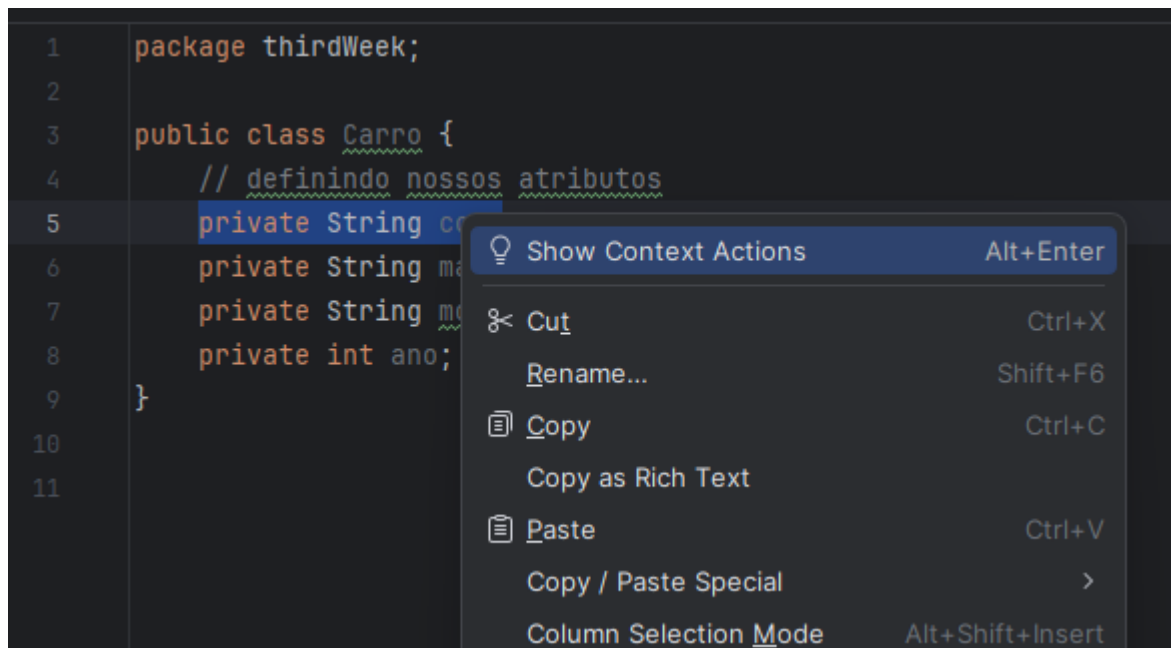
Com a opção escolhida, teremos um novo menu agora para escolher quais atributos desejamos criar os métodos getters and setters de forma automática. Neste caso, e quase sempre, iremos marcar todos:



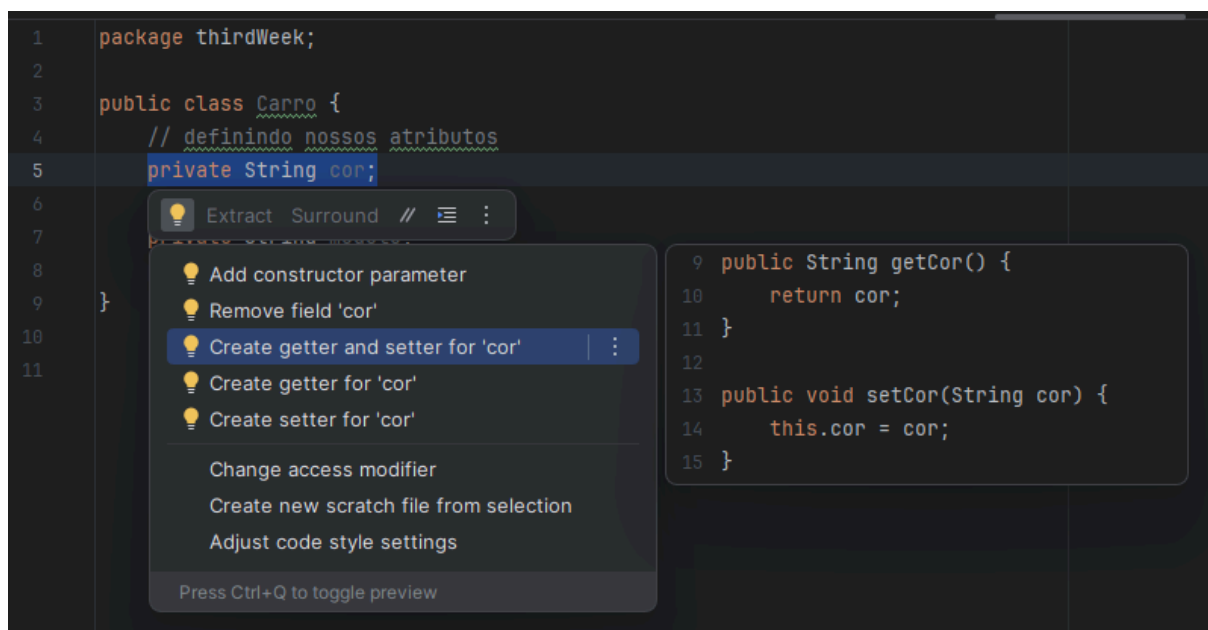
Com isso feito, basta clicar em Generate e tudo será feito automaticamente.

Dentro do IntelliJ, IDE que estou utilizando, o processo é semelhante, porém, com alguns menus diferentes. O que mais difere é que, neste IntelliJ que estou usando (não sei se isso inclui outras versões do IntelliJ IDEA) o processo para criar estes métodos é selecionando atributo por atributo.

Vamos selecionar o primeiro atributo, clicar com o botão direito e selecionar a opção Show Context Actions:



Ao fazer isso, o seguinte menu irá abrir e iremos selecionar a opção que está indicada em azul (Create getter and setter for 'cor'):



Infelizmente, esse processo terá que ser repetido para cada um dos atributos presentes na sua classe.

Uma vez que este processo foi feito para todos os atributos, seu código agora está protegido e de acordo com a orientação a objetos.

Mas, não para por aí. Nós temos outro carinha muito importante dentro da nossa classe. Estamos falando do método construtor. Este método serve como um molde para quando formos inicializar nosso objeto dentro do código, ou seja, criar uma instância. No nosso exemplo da classe carro, nós estamos apenas criando a classe e ela está sem uso nenhum. Bom, o uso da classe será através da criação de um objeto, que nada mais é do que o ato de instanciar nossa classe. Fazer isso é bem simples, veja (agora estaremos criando isso dentro de uma main):

```
Java
public class Teste {
    public static void main(String[] args) {
        Carro c1 = new Carro();
    }
}
```

Vamos analisar a sintaxe:

Carro c1 = new Carro();

O processo é muito semelhante ao de criar uma variável. No exemplo acima, está sendo criado um objeto (parte azul) chamado de c1. Esse objeto é do tipo Carro (indicado em vermelho). O trecho depois do sinal de igual serve para criar o nosso objeto em si, onde temos a palavra reservada new e a invocação do nosso método construtor. O nosso método construtor pode ser vazio, ou então, com os atributos que você desejar. Vamos verificar como criar o método construtor dentro do nosso código:

```
Java
package thirdWeek;

public class Carro {
    // definindo nossos atributos
    private String cor;
    private String marca;
    private String modelo;
    private int ano;
}
```

```
// criando o construtor vazio
public Carro() {}
}
```

Observe que a sintaxe para criar um construtor é muito simples. Nós estamos criando um método público que recebe o mesmo nome da classe em questão (no nosso exemplo é a classe carro). Dentro dos parênteses, você pode passar os atributos que você desejar. Por exemplo, se você quer iniciar um objeto do tipo carro já com um ano de fabricação, uma marca e um modelo, o seu construtor ficaria da seguinte forma:

```
Java
package thirdWeek;

public class Carro {
    // definindo nossos atributos
    private String cor;
    private String marca;
    private String modelo;
    private int ano;


    // definindo o construtor vazio
    public Carro() {}

    // definindo o construtor com argumentos
    public Carro(String marca, String modelo, int ano) {
        this.marca = marca;
        this.modelo = modelo;
        this.ano = ano;
    }
}
```

Foi criado, agora, um novo método construtor com o mesmo nome do construtor vazio (não importa quantos construtores você criar, todos eles receberão o nome da classe), porém, agora ele está recebendo 3 parâmetros: marca, modelo e ano. Quando o objeto for criado através do construtor, os valores que vierem por parâmetro serão associados aos atributos da classe. Isso ocorre por conta da palavra this. Essa palavra indica que estamos nos referindo aos atributos daquela classe, e não às variáveis que estão vindo via parâmetro.

Na imagem a seguir, podemos ver melhor esse funcionamento. Imagine que o primeiro trecho, que refere-se ao ato de instanciar nosso objeto chamado c2, está sendo feito dentro da main, e a criação do nosso construtor - claramente - está sendo feito dentro da nossa classe Carro:

```
Carro c2 = new Carro("Honda", "HRV", 2024)
```



```
public Carro(String marca, String modelo, int ano) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.ano = ano;  
}
```

Perceba para onde vão as setas. Tudo o que está sendo passado na chamada do nosso método construtor na criação do objeto, será substituído, por baixo dos panos, dentro da nossa classe.

Em outras palavras, o que está sendo feito é - basicamente - o seguinte:

```
public Carro("Honda", "HRV", 2024) {  
    this.marca = "Honda";  
    this.modelo = "HRV";  
    this.ano = 2024;  
}
```

Sinta-se livre para criar quantos construtores você quiser. Como todos recebem o mesmo nome, você apenas deve se atentar para os parâmetros. Um método irá se diferenciar do outro através dos parâmetros, caso você crie dois métodos construtores com os mesmos parâmetros resultará em um erro. Como último exemplo, vamos criar um novo construtor que irá receber todos os atributos da classe:

```

Java
package thirdWeek;

public class Carro {
    // definindo nossos atributos
    private String cor;
    private String marca;
    private String modelo;
    private int ano;

    // definindo o construtor vazio
    public Carro() {}

    // definindo o construtor com argumentos
    public Carro(String marca, String modelo, int ano) {
        this.marca = marca;
        this.modelo = modelo;
        this.ano = ano;
    }

    // definindo o construtor com todos os atributos da classe
    public Carro(String cor, String marca, String modelo, int ano) {
        this.cor = cor;
        this.marca = marca;
        this.modelo = modelo;
        this.ano = ano;
    }
}

```

Vamos para dentro da nossa main e instanciar alguns objetos do tipo Carro:

```

Java
package exemplo;

public class Exemplo {
    public static void main(String[] args) {
        // criando objeto a partir do construtor vazio
        Carro c1 = new Carro();

        // criando objeto a partir do construtor que contém marca,
        // modelo e ano
        Carro c2 = new Carro("Honda", "HRV", 2024);

        // criando objeto a partir do construtor que contém cor, marca,
        // modelo e ano
        Carro c3 = new Carro("Preto", "Honda", "HRV", 2024);
    }
}

```

```
    }  
}
```

Neste momento, nós já temos alguns objetos criados dentro do nosso código e a nossa classe Carro ficará assim (com construtores e métodos de acesso getters e setters):

```
Java  
package thirdWeek;  
  
public class Carro {  
    // definindo nossos atributos  
    private String cor;  
    private String marca;  
    private String modelo;  
    private int ano;  
  
    // definindo o construtor vazio  
    public Carro() {}  
  
    // definindo o construtor com argumentos  
    public Carro(String marca, String modelo, int ano) {  
        this.marca = marca;  
        this.modelo = modelo;  
        this.ano = ano;  
    }  
  
    // definindo o construtor com todos os atributos da classe  
    public Carro(String cor, String marca, String modelo, int ano) {  
        this.cor = cor;  
        this.marca = marca;  
        this.modelo = modelo;  
        this.ano = ano;  
    }  
  
    // método get para o atributo cor  
    public String getCor() {  
        return cor;  
    }  
  
    // método set para o atributo cor  
    public void setCor(String cor) {  
        this.cor = cor;  
    }  
}
```

```

// método get para o atributo marca
public String getMarca() {
    return marca;
}

// método set para o atributo marca
public void setMarca(String marca) {
    this.marca = marca;
}

// método get para o atributo modelo
public String getModelo() {
    return modelo;
}

// método set para o atributo modelo
public void setModelo(String modelo) {
    this.modelo = modelo;
}

// método get para o atributo ano
public int getAno() {
    return ano;
}

// método set para o atributo ano
public void setAno(int ano) {
    this.ano = ano;
}
}

```

Com a nossa classe pronta, vamos começar a acessar as informações que já temos e adicionar novos dados aos objetos. Como o objeto c3 já está preenchido com todos os atributos, vamos pedir para o nosso código imprimir as informações desse objeto:

```

Java
package thirdWeek;

public class Testing {
    public static void main(String[] args) {
        // criando objeto a partir do construtor vazio
    }
}

```

```

        Carro c1 = new Carro();

        // criando objeto a partir do construtor que contém marca, modelo e
ano
        Carro c2 = new Carro("Honda", "HRV", 2024);

        // criando objeto a partir do construtor que contém cor, marca,
modelo e ano
        Carro c3 = new Carro("Preto", "Honda", "HRV", 2024);

        System.out.println("----- ESPECIFICAÇÕES DO CARRO -----");
        System.out.println("Cor: " + c3.getCor());
        System.out.println("Marca: " + c3.getMarca());
        System.out.println("Modelo: " + c3.getModelo());
        System.out.println("Ano: " + c3.getAno());
    }
}

```

Ao executar o código acima, temos como resultado:

```

Unset
----- ESPECIFICAÇÕES DO CARRO -----
Cor: Preto
Marca: Honda
Modelo: HRV
Ano: 2024

```

Analise bem o que está ocorrendo. Dentro dos nossos métodos de impressão `System.out.println`, estamos acessando os métodos que o nosso objeto `c3` possui. Ao digitar o nome do objeto e, em seguida, teclar `.` (ponto), todos os métodos presentes da nossa classe `Carro` tornam-se visíveis na IDE para autocompletar o nosso código. Ou seja, ao digitar `c3.` eu estou dizendo que estou dentro do objeto e tenho acesso a todas as funcionalidades que estão descritas como public. Neste caso, como queremos recuperar qual a cor, marca, modelo e ano do nosso objeto `c3`, nós acessamos os métodos `get`, os quais retornam essa informação para nós. Por outro lado, vamos pensar no nosso objeto `c1` que, a princípio, foi criado vazio. Vamos colocar alguns valores para esse objeto? Para isso, vamos utilizar os métodos `set`:


```

Java
package thirdWeek;

public class Testing {
    public static void main(String[] args) {
        // criando objeto a partir do construtor vazio
        Carro c1 = new Carro();

        // criando objeto a partir do construtor que contém marca, modelo e
ano
        Carro c2 = new Carro("Honda", "HRV", 2024);

        // criando objeto a partir do construtor que contém cor, marca,
modelo e ano
        Carro c3 = new Carro("Preto", "Honda", "HRV", 2024);

        c1.setCor("Branco");
        c1.setMarca("Mitsubishi");
        c1.setModelo("Lancer EVO");
        c1.setAno(2020);

        System.out.println("----- ESPECIFICAÇÕES DO CARRO -----");
        System.out.println("Cor: " + c1.getCor());
        System.out.println("Marca: " + c1.getMarca());
        System.out.println("Modelo: " + c1.getModelo());
        System.out.println("Ano: " + c1.getAno());
    }
}

```

Note que, agora, os nossos métodos de impressão estão relacionados com o objeto c1, o qual acabamos de definir novos valores através do set. Vamos executar o código para ver qual o resultado emitido:

```

Unset
----- ESPECIFICAÇÕES DO CARRO -----
Cor: Branco
Marca: Mitsubishi
Modelo: Lancer EVO
Ano: 2020

```

Como esperado, agora nosso objeto c1 possui todos os valores que foram atribuídos para ele através dos métodos set. Vamos comentar todos os métodos set:

```

Java
package thirdWeek;

public class Testing {
    public static void main(String[] args) {
        // criando objeto a partir do construtor vazio
        Carro c1 = new Carro();

        // criando objeto a partir do construtor que contém marca, modelo e
ano
        Carro c2 = new Carro("Honda", "HRV", 2024);

        // criando objeto a partir do construtor que contém cor, marca,
modelo e ano
        Carro c3 = new Carro("Preto", "Honda", "HRV", 2024);

        // c1.setCor("Branco");
        // c1.setMarca("Mitsubishi");
        // c1.setModelo("Lancer EVO");
        // c1.setAno(2020);

        System.out.println("----- ESPECIFICAÇÕES DO CARRO -----");
        System.out.println("Cor: " + c1.getCor());
        System.out.println("Marca: " + c1.getMarca());
        System.out.println("Modelo: " + c1.getModelo());
        System.out.println("Ano: " + c1.getAno());
    }
}

```

E rodar novamente para ver qual o novo resultado:

```

Unset
----- ESPECIFICAÇÕES DO CARRO -----
Cor: null
Marca: null
Modelo: null
Ano: 0

```

Com os métodos comentados, não estamos passando nenhum tipo de valor para o objeto c1. Nesse meio, temos um objeto que está vazio. Dentro do Java, cada tipo de dado tem uma forma de representar o vazio. No caso da String, é através da palavra null e, para o int, é atribuindo 0 à aquela variável.

ArrayList

Até agora você já é capaz de criar variáveis, construir fluxos dentro do seu código e aplicar o paradigma da programação orientada a objetos. Nesse meio, é fato que utilizamos uma variável para guardar algum tipo de informação, mas se for necessário guardar várias informações do mesmo tipo, como isso seria feito? Para lidar com este problema, uma possível abordagem é o ArrayList. Através dessa funcionalidade nós podemos guardar diversas informações que são do mesmo tipo através de uma única estrutura.

Imagine que você deseja guardar 10 números, com o conhecimento anterior você teria que criar 10 variáveis sendo que cada uma dessas 10 variáveis irá conter um valor. Entretanto, isso é totalmente inviável. Com o ArrayList, nós resolvemos este problema de uma forma muito mais simples que consiste na criação de uma variável que é do tipo ArrayList e essa variável simula uma lista, onde você pode exercer várias funções, como adicionar ou remover algum item. Vamos verificar qual é a estrutura básica dessa estrutura de dados:

```
ArrayList<Integer> numeros = new ArrayList<Integer>();
```

Para criar uma ArrayList, nós devemos, primeiramente, definir qual o tipo de dado que essa lista vai conter. Por exemplo, você quer armazenar números inteiros, quebrados, textos, booleanos... e, como observação, para definir o tipo de uma ArrayList, você deve usar tipos não primitivos. Um tipo primitivo é um tipo de dado definido pela própria linguagem para representar um valor usual e simples. Como exemplos de tipos primitivos no Java, nós temos: int, double, float, char, boolean...

O “tipo” String, na real, não é um tipo. Ele é uma classe. Então, quando você for definir uma ArrayList que contenha textos, você pode utilizar o String sem problemas. No exemplo acima, como iremos representar uma lista de números inteiros, foi preciso utilizar um outro tipo para representar o int que, nesse caso, é o Integer. Esse “outro tipo” é chamado de Wrapper Class. Basicamente, uma Wrapper Class permite você interpretar um tipo de dado como um objeto, dessa forma torna-se possível a utilização de métodos sobre os dados que estão armazenados na variável.

Para saber mais sobre Wrapper Classes, acesse o link abaixo:

https://www.w3schools.com/java/java_wrapper_classes.asp

Uma vez que temos o tipo de dado (indicado em vermelho) que a nossa lista irá armazenar definido, é necessário definir qual será o nome da lista (indicado em verde). Afinal, nós iremos utilizar o nome para acessar todos os métodos de manipulação permitidos.

Até o momento, nosso código está ficando assim:

```
Java
package thirdWeek;

import java.util.ArrayList;

public class Testing {
    public static void main(String[] args) {
        ArrayList<Integer> numeros = new ArrayList<Integer>();
    }
}
```

A lista está vazia, vamos adicionar alguns elementos para ela? Para isso, vamos utilizar o método add() que está presente dentro da classe ArrayList:

```
Java
package thirdWeek;

import java.util.ArrayList;

public class Testing {
    public static void main(String[] args) {
        // criando a lista
        ArrayList<Integer> numeros = new ArrayList<Integer>();

        // adicionando elementos para a lista
        numeros.add(1);
        numeros.add(2);
        numeros.add(3);
    }
}
```

Bem fácil, concorda? Vamos criar um laço de repetição for para percorrer sobre todos os itens da nossa lista e imprimi-los em nossa tela:

```

Java
package thirdWeek;

import java.util.ArrayList;

public class Testing {
    public static void main(String[] args) {
        // criando a lista
        ArrayList<Integer> numeros = new ArrayList<Integer>();

        // adicionando elementos para a lista
        numeros.add(1);
        numeros.add(2);
        numeros.add(3);

        // percorrendo por todos os itens presentes em nossa lista
        for(Integer numero : numeros) {
            System.out.println(numero);
        }
    }
}

```

Como resultado, obtemos:

```

Unset
1
2
3

```

Esse for parece diferente do for que eu passei alguns tópicos acima, certo? Bom, o seu funcionamento é praticamente o mesmo, ele só funciona de uma forma mais enxuta. Basicamente, nós criamos uma variável e, para cada repetição, essa variável vai assumir o valor na posição atual da lista. Por exemplo, na primeira iteração, a variável número vai assumir o valor que estiver na primeira posição na nossa lista, lembrando que, dentro da programação, a primeira posição/índice sempre é o número zero. Para visualizar isso, veja a imagem abaixo:



No exemplo, a variável iria assumir o valor 1. Ao passar para a próxima iteração, a variável número vai assumir o valor 2 e assim por diante. Na imagem a seguir, você pode verificar a estrutura deste for de uma forma mais simplificada:

```
for(Integer numero : numeros) {
```

← Tipo de dado da variável que iremos criar para assumir os valores presentes na lista

Nome da variável

Nome da lista que iremos percorrer

Neste contexto, já conseguimos adicionar elementos dentro da nossa lista e imprimir esses elementos na nossa tela. Agora, vamos verificar como é possível remover algum elemento que já foi inserido:

```
Java
package thirdWeek;

import java.util.ArrayList;

public class Testing {
    public static void main(String[] args) {
        // criando a lista
        ArrayList<Integer> numeros = new ArrayList<Integer>();

        // adicionando elementos para a lista
        numeros.add(1);
        numeros.add(2);
        numeros.add(3);

        // percorrendo por todos os itens presentes em nossa lista
        for(Integer numero : numeros) {
            System.out.println(numero);
        }
    }
}
```

```

        // removendo o elemento presente no índice 0
        numeros.remove(0);

        // percorrendo por todos os itens presentes em nossa lista
        for(Integer numero : numeros) {
            System.out.println(numero);
        }
    }
}

```

Para remover um item, foi utilizado o método `remove`. Este método recebe um parâmetro que refere-se à qual índice está o número que desejamos remover. Neste caso, foi passado o índice 0, dessa forma, o número que será removido da lista será o número 1. Vamos executar o código e analisar a saída:

```

Unset
1
2
3
2
3

```

Como esperado, na segunda execução do laço de repetição que está após o `numeros.remove(0)`, o número 1 não está sendo impresso mais, o que corrobora para identificarmos que, de fato, o número 1 foi removido da nossa lista.

A classe `ArrayList`, que representa uma estrutura de dados, possui diversos métodos para realizar a manipulação da lista. Com isso em mente, não entrarei em detalhes de todos os métodos, portanto, para saber mais, acesse o link abaixo:

https://www.w3schools.com/java/java_arraylist.asp