TOMADA DE DECISÃO

Dentro de um bloco, podemos declarar variáveis e usá-las.
Em Java, toda variável tem um tipo que não pode ser mudado, uma vez que declarado:

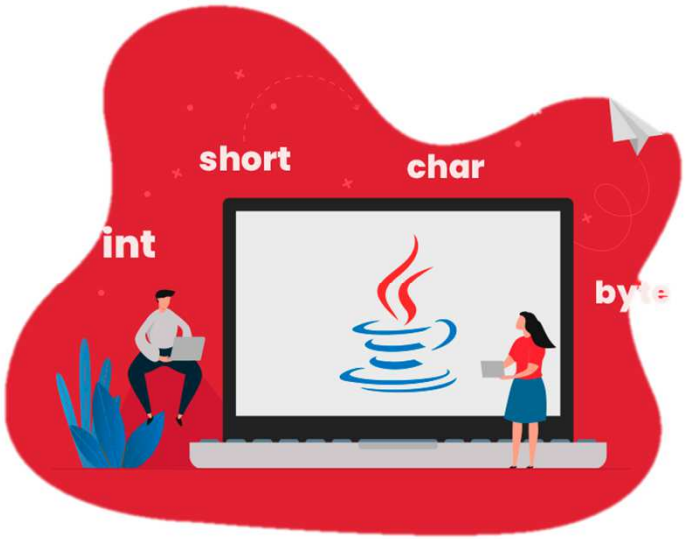
tipoDaVariavel nomeDaVariavel;

Por exemplo, é possível ter uma idade que vale um número inteiro:

int idade;

Com isso, você declara a variável idade, que passa a existir a partir deste momento. Ela é do tipo int, que guarda um número inteiro. A partir de agora, você pode usá-la, primeiro atribuindo valores.
A linha a seguir é a tradução de: “idade deve valer agora quinze”.

idade = 15;



TOMADA DE DECISÃO

Representar números inteiros é fácil, mas como guardar valores reais, tais como frações de números inteiros e outros? Outro tipo de variável muito utilizado é o double, que armazena um número com ponto flutuante (e que também pode armazenar um número inteiro).

```
double pi = 3.14;
double x = 5 * 10;
```

O tipo boolean armazena um valor verdadeiro ou falso, e só: nada de números, palavras ou endereços, como em algumas outras linguagens.

```
boolean verdade = true;
```

true e false são palavras do Java. É comum que um boolean seja determinado através de uma **expressão booleana**, isto é, um trecho de código que retorna um booleano, como o exemplo:

```
int idade = 30;
boolean menorDeIdade = idade < 18;
```

O tipo char guarda um, e apenas um, caractere. Esse caractere deve estar entre aspas simples. Não se esqueça dessas duas características de uma variável do tipo char! Por exemplo, ela não pode guardar um código como ” pois o vazio não é um caractere!

```
char letra = ’a’;
```

TOMADA DE DECISÃO

Alguns valores são incompatíveis se você tentar fazer uma atribuição direta. Enquanto um número real costuma ser representado em uma variável do tipo double, tentar atribuir ele a uma variável int não funciona porque é um código que diz: **"i deve valer d"**, mas não se sabe se d realmente é um número inteiro ou não.

```
double d = 3.1415;
int i = d; // não compila
```

O mesmo ocorre no seguinte trecho:

```
int i = 3.14;
```

O mais interessante, é que nem mesmo o seguinte código compila:

```
double d = 5; // ok, o double pode conter um número inteiroint i = d; // não
compila
```

Apesar de 5 ser um bom valor para um int, o compilador não tem como saber que valor estará dentro desse double no momento da execução. Esse valor pode ter sido digitado pelo usuário, e ninguém vai garantir que essa conversão ocorra sem perda de valores.

TOMADA DE DECISÃO

Já no caso a seguir, é o contrário:

```
int i = 5; double d2 = i;
```

O código acima compila sem problemas, já que um double pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo int podem ser guardados em uma variável double, então não existem problemas no código acima.

Às vezes, precisamos que um número quebrado seja arredondado e armazenado num número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja **moldado (casted)** como um número inteiro. Esse processo recebe o nome de **casting**.

```
double d3 = 3.14;int i = (int) d3;
```

O casting foi feito para moldar a variável d3 como um int. O valor de i agora é 3. O mesmo

ocorre entre valores int e long.

```
long x = 10000;
int i = x; // não compila, pois pode estar perdendo informação
```

E, se quisermos realmente fazer isso, fazemos o casting:

```
long x = 10000; int i = (int) x;
```

TOMADA DE DECISÃO

Alguns **castings** aparecem também:

```
float x = 0.0;
```

O código acima não compila pois todos os literais com ponto flutuante são considerados double pelo Java. E float não pode receber um double sem perda de informação, para fazer isso funcionar podemos escrever o seguinte:

```
float x = 0.0f;
```

A letra f, que pode ser máiscula ou minúscula, indica que aquele literal deve ser tratado como float. Outro caso, que é mais comum:

```
double d = 5; float f = 3;
float x = f + (float) d;
```

Você precisa do casting porque o Java faz as contas e vai armazenando sempre no maior tipo que apareceu durante as operações, no caso o double.
E, uma observação: no mínimo, o Java armazena o resultado em um int, na hora de fazer as contas.

TOMADA DE DECISÃO

Castings possíveis

Abaixo estão relacionados todos os casts possíveis na linguagem Java, mostrando a conversão **de** um valor **para** outro. A indicação **Impl.** quer dizer que aquele cast é implícito e automático, ou seja, você não precisa indicar o cast explicitamente (lembrando que o tipo boolean não pode ser convertido para nenhum outro tipo).

PARA: / DE:	byte	short	char	int	long	float	double
byte	----	*Impl.*	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
short	(byte)	----	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
char	(byte)	(short)	----	*Impl.*	*Impl.*	*Impl.*	*Impl.*
int	(byte)	(short)	(char)	----	*Impl.*	*Impl.*	*Impl.*
long	(byte)	(short)	(char)	(int)	----	*Impl.*	*Impl.*
float	(byte)	(short)	(char)	(int)	(long)	----	*Impl.*
double	(byte)	(short)	(char)	(int)	(long)	(float)	----

TOMADA DE DECISÃO

Na tabela abaixo, estão os tamanhos de cada tipo primitivo do Java.

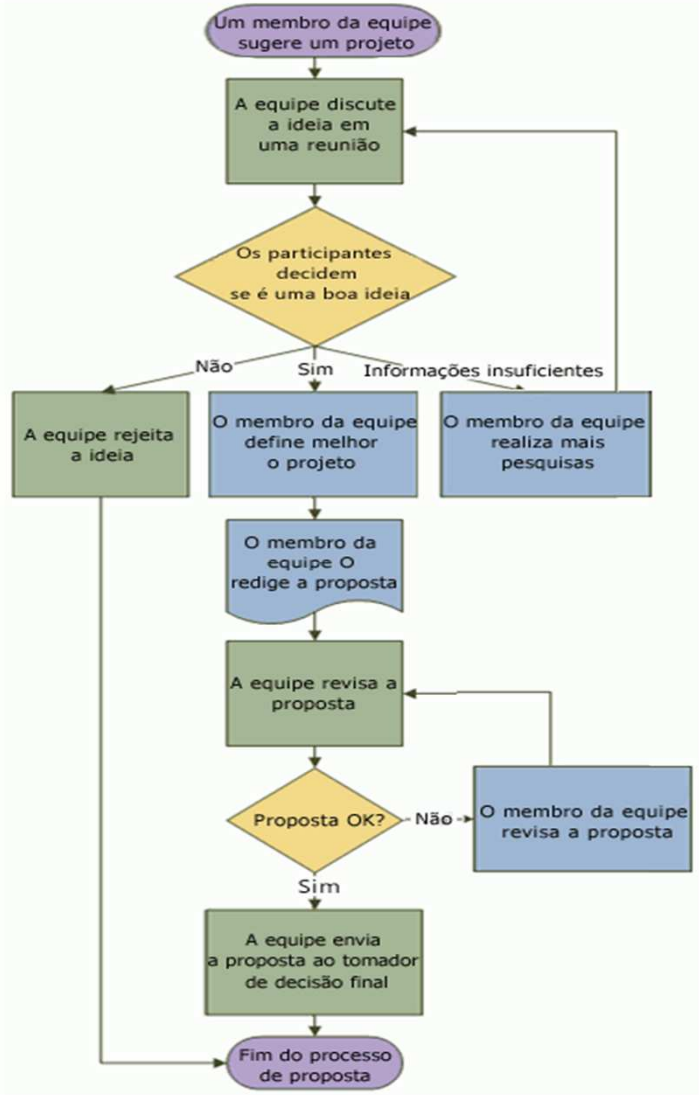
TIPO	***TAMANHO***
boolean	1 bit
byte	1 byte
short	2 bytes
char	2 bytes
int	4 bytes
float	4 bytes
long	8 bytes
double	8 bytes

TOMADA DE DECISÃO

Os fluxogramas são diagramas que mostram as etapas de um processo. Os fluxogramas básicos são fáceis de criar e, como as formas são simples e visuais, são fáceis de entender.

O modelo Fluxograma Básico vem com formas que podem ser usadas para mostrar vários tipos de processos e é especialmente útil para mostrar processos empresariais básicos, como o processo de desenvolvimento de proposta mostrado na figura a seguir.

https://app.diagrams.net/

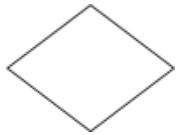


TOMADA DE DECISÃO

- `Inicial/Final` - Use essa forma para a primeira e a última etapa do seu processo.

- `Processo` - Essa forma representa uma etapa típica do seu processo. Esta é a forma usada com mais frequência em quase todos os processos.

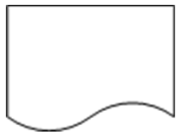
- `Decisão` - Esta forma indica o ponto em que o resultado de uma decisão ditará a próxima etapa. É possível que haja vários resultados, mas normalmente há somente dois: sim e não.



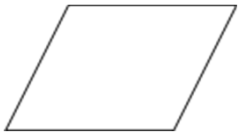
- `Subprocesso` - Use esta forma para um conjunto de etapas que se combinam para criar um subprocesso definido em outro lugar, geralmente em outra página do mesmo documento. Isto será útil sempre que o diagrama for muito longo e complexo.

TOMADA DE DECISÃO

- Documento - Essa forma representa uma etapa que resulta em um documento.



- Dados - Essa forma indica que informações estão entrando no processo a partir do exterior, ou deixando-o. Essa forma também pode ser usada para representar materiais e às vezes é chamada de forma Entrada/Saída.



- Referência na página - Esse pequeno círculo indica que a etapa seguinte (ou anterior) está em algum outro local do desenho. Isso é particularmente útil em fluxogramas grandes, onde você teria que usar um conector longo, que pode ser difícil de acompanhar.

- Referência fora da página - Quando você soltar essa forma na página de desenho, será aberta uma caixa de diálogo onde é possível criar um conjunto de hiperlinks entre duas páginas de um fluxograma ou entre uma forma de subprocesso e uma página separada de fluxograma que mostre as etapas do subprocesso.

TOMADA DE DECISÃO

Tomada de Decisão, ou então, estrutura condicional, é o julgamento de uma condição lógica e a consequente decisão de qual bloco de código que será executado. Ou seja, essa a maneira em que os nossos programas irão tomar decisões baseando-se em condições pré-determinadas.
Utilizamos a instrução if, para verificar se uma condição é ou não verdadeira. Nós temos, que if do inglês, significa se ou então, caso. A instrução if é a responsável por decidir que: se for verdadeiro, execute esse bloco de instrução, do contrário não faça nada ou então, execute este outro bloco de instrução. A seguir, nós temos a estrutura da instrução if:

```
se( condição ){

}

if( condição ){

}
```

Se o valor avaliado é verdadeiro, a execução do programa executara as instruções contidas nas chaves, do contrário, nada acontece.
Fato é, que com a instrução if conseguimos manipular a execução dos nossos código, podendo desviar o fluxo de execução quando uma determinada condição for satisfeita.

TOMADA DE DECISÃO

```
Scanner in = new Scanner(System.in);

System.out.println("Digite um número qualquer: ");
int num = in.nextInt();

if(num > 50){
    System.out.println("O número digitado é maior do que cinquen
ta.");
}
```

No exemplo acima, nós pedimos para que o usuário digitasse um número e em seguida nós verificamos se o número digitado é maior do que 50. Caso a avaliação seja positiva, ou seja, caso o valor contido na variável num realmente seja maior do que 50, o bloco de instrução da instrução **if** será executado.

Assim, nós temos que o funcionamento da instrução **if** se resume a avaliar a expressão contida entre os parêntesis. Se a condição for igual a **true**, ou seja, se o valor retornado for verdadeiro, o bloco de instrução que é definido pelo uso de um par de chaves será executado.

TOMADA DE DECISÃO

A leitura correta da instrução if é:

```
if( condição == true)
```

O sinal de igual, seguido de outro sinal de igual, significa que estamos querendo verificar se o valor a esquerda é igual ao valor da direita.

```
if( 1>1 == true)
```

O exemplo acima deve ser lida da seguinte maneira: o número 1 é maior do que o número 1? Sim ou Não? O valor dessa verificação, então será comparado com o membro a direita do sinal que verifica se ambos valores são iguais.

TOMADA DE DECISÃO

É muito comum, os programadores omitirem o [== true], até porque, a condição if sempre ira verificar se o valor é verdadeiro. Ou seja, toda expressão colocada entre os parêntesis esta, implicitamente, sendo comparada a true. Como estudaremos, nós podemos inverter esse funcionamento, mas o princípio nunca será alterado. Assim, temos que as duas expressões a seguir estão sendo comparadas com o valor true - estão sendo verificadas se são ou não verdadeiras:

```
//o membro a esquerda do sinal de comparação (1>1)
//está sendo comparado ao valor a direita do sinal
//de igualdade (true).
//Podemos dizer que [(1>1) é igual a (true)].
if( 1>1 == true)

//é verdadeira a expressão contida nos parêntesis?
if( 1>1)

//o membro
if( 1>1 == true)
```

EXEMPLO A SEGUIR

```
import java.util.Scanner;

public class Aula0010 {
    public static void main(String[] args) {
//          int num;
//          num = 11;
//
//          if(num == 10){
//              System.out.println("sim, é igual");
//          }else{
//              System.out.println("não, o número não é");
//
//          }

          int num;
          System.out.println("Digite o número 1: ");
          Scanner in = new Scanner( System.in );
          num = in.nextInt();
          if(num==1){
              System.out.println("Obrigado por digitar o número 1"
);
          }else{
              System.out.println("O número digitado não é igual a
1");
          }
    }
}
```

TOMADA DE DECISÃO

Aprendemos, que com a tomada de decisão, somos capazes de avaliar uma condição e a partir desta, decidir se um bloco de instrução será executado.
Agora, vamos estudar a situação em que a condição não é verdadeira, ou seja, vamos estudar o que deve acontecer nas situações em o valor lógico for igual a false.
Nós definimos uma expressão, ou melhor, nós definimos uma condição. Nós temos, que o interpretador pegara a expressão contida entre os parêntesis e a substituirá por um valor booleano - true/false. Se o valor substituído for true as instruções representadas pelo retângulo serão executadas, do contrário, nada ocorrerá e o nosso programa continuará a sua execução normalmente.

Em praticamente toda entrada de dado nós devemos verificar se o valor informado corresponde ao valor esperado. Isso porque, em um cadastro de pessoas não podemos permitir que o campo idade contenha letras. Assim, antes de enviar para o banco de dados, nós devemos verificar todas as informações, e caso alguma não esteja correta, temos que levantar uma mensagem de erro para que a informação incorreta seja corrigida.

TOMADA DE DECISÃO

Quando um programa exibe uma mensagem de erro, trava ou fecha abruptamente, o que aconteceu foi uma situação inesperada e caso houvesse uma tomada de decisão, ocorreria um desvio na execução das instruções e o programa não apresentaria esse bug.

Nós podemos dizer que um bug em um programa é o resultado da não verificação de um determinado estado. Quando nosso programa não está preparado para uma situação, a execução deste será aleatória e muito provavelmente será exibido uma mensagem de erro. Porém, há diversos erros em que nenhuma mensagem é apresentada, como por exemplo, um usuário informar que seu nome é um número com caracteres especiais.

Se o nosso programa não verificar todos os caracteres digitados, o nosso programa terá um erro e este não levantara nenhuma mensagem de erro, e isso irá causar uma série de informações inválidas no cadastro. Logo, pela falta de uma verificação, acabamos com uma base de dados que contém informações erradas. Esse é um outro grupo de erro e geralmente, quando percebemos, há muita informação desconexa e essas deveram ser corrigidas.

TOMADA DE DECISÃO

Toda vez que você deixa seu programa causar um erro na cara do usuário e fechar,

Um bebê foca morre...

E você não quer isso

Trate possíveis erros, os bebês focas agradecem!

TOMADA DE DECISÃO

Agora vamos fazer um programa para praticarmos tudo que já estudamos:

```
Scanner in = new Scanner(System.in);

System.out.println("Digite '1' para sim e digite '0' para não: "
);
int acao = in.nextInt();

if(acao==1){
    print("Você disse que sim!")
}

if(acao==0){
    print("Você disse que não!")
}
```

No código acima, criamos uma instância de Scanner e atribuímos a mesma para a variável in. Em seguida, escrevemos uma mensagem para o usuário pedindo para que o mesmo digite 1 para dizer que sim, e 0 para dizer que não.

TOMADA DE DECISÃO

Em seguida, nós verificamos com a instrução if se o número digitado pelo usuário é igual a 1. Se sim, escrevemos na tela uma mensagem confirmando a operação.
Em seguida, nós avaliaremos outra expressão com a instrução if, se o número contido na variável for igual a 0, imprimimos uma mensagem para o usuário confirmando o valor digitado.

O programa que acabamos de fazer funciona perfeitamente e nós conseguimos o resultado esperado. Porém, estamos utilizando duas estruturas if, enquanto poderíamos utilizar somente uma. E também, caso a condição da expressão contida no primeiro if seja verdadeiro, não há necessidade para verificarmos novamente o valor contido na variável ação, até porque, já conhecemos.
Esse exemplo teve como objetivo, mostrar a necessidade de combinar outras instruções junto com a tomada de decisão if. Mesmo que o resultado seja o esperado, nosso código, mesmo que correto possui erros lógicos.

ELSE

A instrução else é utilizada para definir o bloco de instrução que deve ser executado quando a condição não for satisfeita.
A palavra else significa em português senão, logo, como o nome sugere, é a instrução que irá conter o que dever ser feito se quando a expressão verificada for igual a false.
A seguir temos um exemplo mostrando a utilização da instrução else:

```
if(false){
}else{
    //bloco que será executado quando a condição não for satisfe
ita
}
```

ELSE

É uma pratica comum a verificação de uma outra condição dentro do bloco else, por exemplo:

```
if(false){
}else{
    if(true){
        //este bloco será executado se a instrução for verdadeir
a
    }
```

No exemplo acima, nós estamos verificando uma condição e definindo dois blocos, um para caso a condição seja atendida e o outro para quando o valor retornado for igual a false. Dentro do bloco de instrução else foi utilizado uma outra tomada de decisão, logo, nós temos que o bloco de instrução da tomada de decisão que está contido na instrução else da primeira tomada de decisão só será executado caso a primeira condição não seja verificada e caso a segundo condição retorne true.

EXEMPLO A SEGUIR

```
public class Aula0011 {
    public static void main(String[] args) {
        //igual
        //diferente

//          System.out.println(1==1);

//          boolean b = 1==1;
//          System.out.println( b );

//          boolean b = 2!=1;
//          System.out.println( b );

//          boolean b = 2>1;

//          System.out.println( b );

        int num1 = 3;
        int num2 = 3;
        System.out.println( num1 > num2 );
        System.out.println( num1 >= num2 );

    }
}
```