



CAPACITAÇÕES  
**JAVA PARA  
A PDPJ-Br**

# JAVA BÁSICO

E-BOOK 2

*Ronaldo Pinheiro Gonçalves Junior*

## 4 Programação orientada a objetos básica

### 4.1 Classes e objetos

Para construir programas utilizando o paradigma orientado a objetos, como na linguagem Java, precisamos entender quais são as formas apropriadas de se representar o mundo real e suas entidades. É importante definir nossos primeiros objetivos básicos ao construir programas de computador (DEITEL, 2016): identificar quais entidades devem estar presentes no programa e como representá-las.

Devemos incluir todo o necessário para a execução do nosso programa, mas precisamos também ficar atentos para não pecar por excesso (MARTIN, 2020). Isto é, não devemos incluir o que é desnecessário. Por exemplo, no contexto de registros de casos judiciais, nosso programa precisa guardar informações de casos, processar essas informações e assim por diante. Logo, a primeira e mais intuitiva entidade que podemos pensar é um *Caso*.

É possível incluir outras entidades, como *Advogado*, *Parte Interessada*, *Julgamento*, entre outras. Note que representar o *Veículo* (*Carro*, *Ônibus* etc.) que o advogado usa para chegar ao tribunal pode não ser importante. Então, enquanto não houver motivação para incluí-la, optamos em deixá-la fora do grupo de entidades do sistema. Essas entidades são chamadas, em Java, de classes, modelos que definem uma estrutura de representação. Uma classe é criada usando a palavra reservada *class*, seguida do nome da classe e duas chaves definem o que está dentro da classe. Por exemplo, para criar uma classe *Caso* (inicialmente “vazia”), que representa um caso judicial, escrevemos a seguinte linha, dentro de um novo arquivo *Caso.java*:

```
class Caso { }
```

Uma vez que temos uma classe definida, podemos instanciar essa classe. O processo de instanciação é a criação de um exemplo concreto da classe em consideração. Chamamos essas instâncias de objetos. Vale ressaltar que classes são estruturas, enquanto objetos são exemplos em execução. Veja, a seguir, um exemplo de instanciação da classe *Caso*. De forma semelhante a uma variável para tipos primitivos, criamos aqui uma variável do tipo *Caso*.

```
Caso casoAtual = new Caso();
```

Múltiplos objetos podem ser instanciados a partir de uma mesma classe, pois a classe é a forma como uma entidade é representada em sua estrutura, enquanto um objeto pode ser visto como uma classe em execução, ou um exemplo concreto dessa estrutura. Nas próximas

seções, veremos como especificar melhor os detalhes de uma classe e como instanciar objetos com dados concretos.

## 4.2 Atributos

Podemos dizer que cada objeto distinto de uma classe *Caso* faz referência a um caso distinto no mundo real. Todavia, cada caso possui atributos próprios, como número, decisão, descrição etc. Para criar um atributo em uma classe Java, utilizamos declarações da mesma forma que fizemos com variáveis de tipo primitivo. O número de um caso pode ser do tipo *int*, enquanto a decisão pode ser um *char*. Ao colocar esses atributos dentro de uma classe, dizemos que eles pertencem a essa classe. Observe, a seguir, uma atualização da classe *Caso*, agora com atributos.

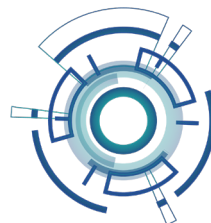
```
class Caso {  
    int numero;  
    char decisao;  
    String descricao;  
}
```

Note que, ao instanciar um *Caso*, utilizando essa nova versão da classe, cada objeto poderá armazenar um valor distinto para número. O mesmo pode ser feito para decisão e descrição.

## 4.3 Métodos

Acabamos de ver que atributos são propriedades de uma classe, como o número de um caso. Essas características de uma classe poderão apresentar valores distintos para cada objeto da classe. Em adição aos atributos, outro conceito importante de classes é que podem definir ações. Isto é, ao executar um programa, podemos utilizar um objeto de uma classe para executar uma ação definida naquela classe. Cada ação é representada por um método, que pode ser executado através de uma chamada. Em outras palavras, ao realizar uma chamada de um método de um objeto, desejamos executar esta ação do objeto.

Ao realizar uma chamada de um método, é possível passar parâmetros, valores de entrada para execução desse método. Podemos ainda receber um retorno do método, o resultado de sua execução. Chamamos de assinatura de um método a definição do tipo de retorno, o nome do método e seus parâmetros. Um exemplo básico que pode ser construído na classe *Caso* é um método para verificar se um caso já tem um veredito:



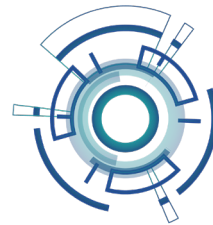
```
boolean temVeredito() {  
    switch (decisao) {  
        case 'A':  
            return true;  
        case 'I':  
            return true;  
        case 'C':  
            return true;  
        default:  
            return false;  
    }  
}
```

O exemplo apresentado mostra o método *temVeredito*, criado dentro da classe *Caso*. Note que a primeira palavra colocada antes do nome do método indica o tipo de retorno do método. Isto é, ao fazer uma chamada ao método *temVeredito*, receberemos como retorno um tipo primitivo booleano, *true* ou *false*. Após o nome do método, temos parênteses para receber parâmetros de entrada, variáveis que podem ser usadas pelo método. Neste caso, não temos nenhum parâmetro, mas em breve veremos exemplos de métodos que usam parâmetros. Dentro do bloco do método *temVeredito*, a primeira linha é uma instrução *switch...case*, que checa o valor do atributo *decisão* de um caso. Se o valor armazenado na *decisão* for *A*, o caso tem o veredito de absolvição. Já um valor *I* indica indeferimento e um valor *C* indica condenação. Nesses três casos, podemos dizer que temos um veredito, então retornamos *true*. Do contrário, se o atributo não tem nenhum desses valores, significa que o caso ainda não tem veredito – retornamos *false*.

#### 4.4 Palavra reservada *this*

Em Java, palavras reservadas são aquelas que não podem ser utilizadas em nomes de variáveis, pois são utilizadas pela própria linguagem. Além dos tipos primitivos e das classes da linguagem Java que vimos anteriormente, outra palavra reservada é o *this*, que em português significa “isto”. A palavra *this* é utilizada, dentro de uma classe, para referenciar diretamente um atributo do objeto que está em execução.

Para melhor explicar seu uso, vamos ressaltar que podemos ter múltiplos objetos de uma única classe, em execução, ao mesmo tempo. Todavia, quando queremos saber o valor de um atributo de um objeto em específico, podemos usar a palavra reservada *this*. Um bom exemplo de uso desta palavra é em um construtor, que veremos a seguir.



## 4.5 Construtores

Uma forma apropriada de se criarem instâncias de uma classe é através de construtores. Dessa forma, podemos inicializar os valores dos atributos da classe para cada novo objeto criado. Um construtor leva o mesmo nome da classe à qual pertence e os valores de inicialização são recebidos como parâmetros. Veja, a seguir, o exemplo de um construtor para a classe *Caso*. Neste exemplo, o construtor possui três parâmetros, um para cada atributo da classe: o primeiro parâmetro é do tipo *int* e é denominado *numero*; o segundo é *decisao*, do tipo *char*; e o terceiro e último é a *descricao*, do tipo *String*.

```
class Caso {
    int numero;
    char decisao;
    String descricao;

    Caso(int numero, char decisao, String descricao) {
        this.numero = numero;
        this.decisao = decisao;
        this.descricao = descricao;
    }

    boolean temVeredito() {
        switch (decisao) {
            case 'A':
                return true;
            case 'I':
                return true;
            case 'C':
                return true;
            default:
                return false;
        }
    }
}
```

Uma dúvida que pode surgir é que, ao escrevermos *numero* dentro do construtor, podemos estar fazendo referência ao parâmetro *numero* ou ao atributo *numero*, pois ambos têm a mesma denominação. Em Java, ao escrever a palavra *numero* dentro do construtor, estaremos fazendo referência ao parâmetro. Entretanto, observe que, logo na primeira linha, podemos fazer uso de ambos, por meio da palavra reservada *this*, que faz referência ao atributo. Em outras palavras, na primeira linha do construtor, o atributo *numero* recebe o valor do parâmetro *numero*. O mesmo acontece na segunda e na terceira linha do construtor para *decisao* e *descricao*, respectivamente.

A instanciação pode possuir valores de inicialização por meio do uso de um construtor. Para configurarmos valores específicos, no momento de criação de um objeto, como caso de número “55”, com decisão “Indeferimento” e descrição “Divórcio e Partilha de Bens”, usamos um construtor da seguinte maneira:

```
Caso casoAtual = new Caso(55, 'I', "Divórcio e Partilha de Bens");
```

Ao executar essa linha de instanciação, o construtor da classe *Caso* será chamado e os valores serão colocados nos atributos desse objeto. Este objeto, em específico, terá estes valores em seus atributos e ficará armazenado na variável *casoAtual*.

#### 4.6 Modificadores de acesso a atributos, métodos e construtores

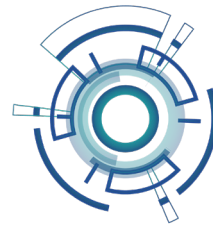
Um importante aspecto para fazer uso de um objeto são os modificadores de acesso. Estes modificadores definem de onde um recurso da linguagem pode ser acessado. Existem três tipos de modificadores de acesso: público, protegido e privado.

Os modificadores podem ser colocados antes de atributos, métodos e construtores para alterar como esses elementos são acessados. Para melhor entender como modificadores funcionam, vamos atualizar nosso exemplo anterior da classe *Caso*, para que ela tenha um atributo público, um atributo protegido e um atributo privado.

```
class Caso {  
    public int numero;  
    protected char decisao;  
    private String descricao;
```

Para o exemplo acima, o atributo *numero* pode ser acessado pela própria classe *Caso* e quaisquer outras classes, pois possui o modificador *public*. Já o atributo *decisao*, com o modificador *protected*, só pode ser acessado pela própria classe *Caso* e classes que estão no mesmo pacote da classe *Caso*. Veremos, em breve, o conceito de herança, mas subclasses também podem acessar atributos *protected*. Por fim, o atributo *descricao* não pode ser acessado de local algum do programa, apenas dentro da classe *Caso*. Quando não utilizamos um modificador, o atributo tem o modificador padrão e só pode ser acessado dentro de um mesmo pacote.

Uma boa prática que utilizamos na construção de programas é declarar, sempre que possível, atributos com o modificador *private*, para que sejam acessados apenas por sua própria classe. Já classes e construtores tendem a ser públicos, para que sejam acessados por outras classes. Nas próximas seções, estudaremos a motivação por trás dessa prática e mostraremos um exemplo atualizado da classe *Caso* com modificadores, a começar pelo encapsulamento de atributos.



## 4.7 Encapsulamento: getters e setters

Conforme vimos anteriormente, é possível acessar um atributo de qualquer classe, utilizando o modificador *public*. Ou seja, outra classe pode alterar o valor do atributo decisão de 'I' para 'A'. Note que existe um problema de segurança ao deixar público o acesso ao atributo. Imagine que uma classe gostaria de mudar o valor de descricao para "A", "55", "Teste" ou qualquer outro valor.

Em geral, devemos proteger a forma com que os atributos de uma classe têm seus valores alterados. Chamamos essa proteção de encapsulamento, através dos métodos *getters* e *setters*. Os métodos *getters*, do inglês "get" (que significa "pegar"), são utilizados para acessar e recuperar os valores de atributos, enquanto os métodos *setters*, do inglês "set" (que significa "colocar"), são utilizados para acessar e alterar os valores de atributos. A menos que haja uma razão para um atributo ser acessado diretamente, todo atributo em Java tende a utilizar o modificador *private* e possuir métodos *getters* e *setters*. Veja como modificamos nossa classe *Caso*, para utilizar os modificadores recomendados, e criamos métodos *getters* e *setters* para o atributo *decisao*.

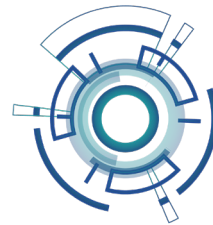
```
public class Caso {
    private int numero;
    private char decisao;
    private String descricao;

    public Caso(int numero, char decisao, String descricao) {
        this.numero = numero;
        this.decisao = decisao;
        this.descricao = descricao;
    }

    public boolean temVeredito() {
        switch (decisao) {
            case 'A':
                return true;
            case 'I':
                return true;
            case 'C':
                return true;
            default:
                return false;
        }
    }

    public char getDecisao() {
        return this.decisao;
    }

    public void setDecisao(char decisao) {
        if(decisao == 'A' || decisao == 'I' || decisao == 'C') {
            this.decisao = decisao;
        }
    }
}
```



A classe *Caso* agora é pública e pode ser utilizada de qualquer parte do programa, mas seus atributos são privados e não podem ser acessados fora da classe. Todavia, podemos acessar o valor da decisão através do método *getDecisao*, que possui modificador público. Esse método de encapsulamento tem um tipo de retorno *char*, o mesmo tipo do atributo *decisao*. Ao realizar uma chamada a esse método, iremos receber o valor que está atualmente armazenado no atributo *decisao*.

O método *setDecisao* também tem modificador de acesso público, para permitir que o valor do atributo *decisao* de um objeto *caso* seja alterado. O tipo de retorno do método é *void*, palavra reservada que é utilizada quando queremos dizer que um método não realizará retorno algum. Finalmente, temos um parâmetro *decisao* do tipo *char*, que carrega o novo valor a ser colocado no atributo *decisao*. Nesse exemplo, estamos encapsulando o atributo *decisao*, permitindo apenas os valores "A", "I" e "C" como novos valores. Caso um valor diferente seja enviado por parâmetro, ele será ignorado, protegendo a modificação do atributo.

## 4.8 Classes aninhadas

Um último conceito da programação orientada a objetos básica que estudaremos agora é o de classes aninhadas. Ao identificar entidades no mundo real e projetá-las como classes em nosso programa, é possível que um programador queira criar uma classe dentro de outra classe. Chamamos esse fenômeno de classes aninhadas.

O uso de classes aninhadas geralmente está associado ao fato de que uma classe interna só é relevante para a classe na qual foi criada. Veremos outros modificadores e versões mais avançadas desse conceito, mas, por ora, vamos observar como seria um exemplo para nossa classe *Caso*:

```
public class Caso {  
    // ...  
    public class Depoimento {  
        private String nomeTestemunha;  
        private String conteudo;  
  
        public Depoimento(String nomeTestemunha, String conteudo) {  
            this.nomeTestemunha = nomeTestemunha;  
            this.conteudo = conteudo;  
        }  
    }  
    // ...  
}
```



A classe *Depoimento* foi criada dentro da classe *Caso*, possui atributos nome da testemunha e conteúdo, e um construtor. Podemos criar objetos da classe *Depoimento*, dentro de objetos da classe *Caso* e os atributos da classe *Caso* são acessíveis pela classe *Depoimento*. O uso de classes aninhadas dessa maneira é recomendado apenas quando utilizamos depoimentos exclusivamente em casos. Em cenários onde isso não acontece, recomenda-se a criação de duas classes distintas, em arquivos diferentes: *Caso.java* e *Depoimento.java*.

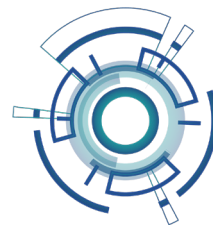
## 5 Operações com Datas e entrada e saída básica de dados

### 5.1 Operações com Datas

Ao utilizar a linguagem Java, algumas classes utilitárias estão disponíveis para facilitar a construção de novos programas, como as classes *String* e *Math*, que estudamos anteriormente. Muitas outras classes estão presentes na linguagem Java e veremos agora algumas que nos ajudam a realizar operações com datas.

Digamos que queremos modificar nossa classe de caso judicial, para ter um atributo de data de abertura. Isto é, precisamos manter o dia, o mês e o ano, em que o caso foi aberto. Seria possível atender a essa demanda, criando um atributo do tipo inteiro, para registrar o dia, um segundo atributo inteiro para registrar o mês e um terceiro atributo inteiro para registrar o ano. Entretanto, toda vez que fôssemos utilizar a data, teríamos que manipular três atributos. Uma forma mais fácil seria utilizar classes como *Date* e *Calendar*, do pacote *java.util*. A partir do Java 8, temos um pacote ainda mais moderno para operações com datas: o pacote *java.time*, com a classe *LocalDate*. Para o exemplo anterior, em vez de usarmos três atributos, podemos simplesmente utilizar a classe *LocalDate*. Em adição, podemos facilmente obter a data atual e armazenar este valor. Veja como ficaria a classe *Caso*, em sua versão atualizada.

```
public class Caso {  
    // ...  
    private LocalDate dataAbertura;  
  
    public Caso(int numero, char decisao, String descricao) {  
        this.numero = numero;  
        this.decisao = decisao;  
        this.descricao = descricao;  
  
        this.dataAbertura = LocalDate.now();  
    }  
    // ...  
}
```



Outras operações comuns que podemos realizar incluem adicionar um determinado número de dias a uma data, subtrair um determinado número de dias de uma data, comparar datas e formatar datas. Observe, a seguir, o uso da classe *LocalDate*, para facilitar essas operações.

```
if(dataAbertura.isEqual(outraData)) {  
    dataAbertura = dataAbertura.plusDays(7);  
    dataAbertura = dataAbertura.minusMonths(2);  
}
```

Nesse trecho de código, verificamos se a data de abertura é a mesma que uma outra data. Caso seja, colocamos a data de abertura 7 dias para frente e, subsequentemente, antecipamos a data de abertura em 2 meses. Além do que foi mostrado no exemplo, temos métodos adicionais para manipulação da data, em que podemos facilmente somar e subtrair dias, meses e anos a uma data. Outras operações de comparação de data estão disponíveis, como *isBefore* e *isAfter*, para saber se uma data vem antes ou depois de outra, respectivamente.

Além de datas, podemos também utilizar períodos por meio da classe *Period* do pacote *java.time*. Essa classe tem o método *between*, que recebe um primeiro parâmetro *LocalDate*, como data de início, e segundo *LocalDate*, como data de fim. Podemos saber quantos dias tem o período por meio do método *getDays*, combinar as classes *LocalDate* e *Period*, para saber se uma data específica está dentro de um período e assim por diante.

Uma última atividade muito utilizada em operações com data é a formatação. Por padrão, uma data do tipo *LocalDate* irá utilizar: quatro dígitos para o ano; um hífen ("-") e dois dígitos para o mês; e um hífen seguido de dois dígitos para o dia. Um exemplo seria "2023-09-07". Para formatar essa data, podemos utilizar a classe *DateTimeFormatter* da seguinte forma:

```
LocalDate dataAbertura = LocalDate.now(); // data atual "2023-09-07"  
DateTimeFormatter formato = DateTimeFormatter.ofPattern("dd/MM/yyyy");  
dataAbertura.format(formato); // produz a String "07/09/2023"
```

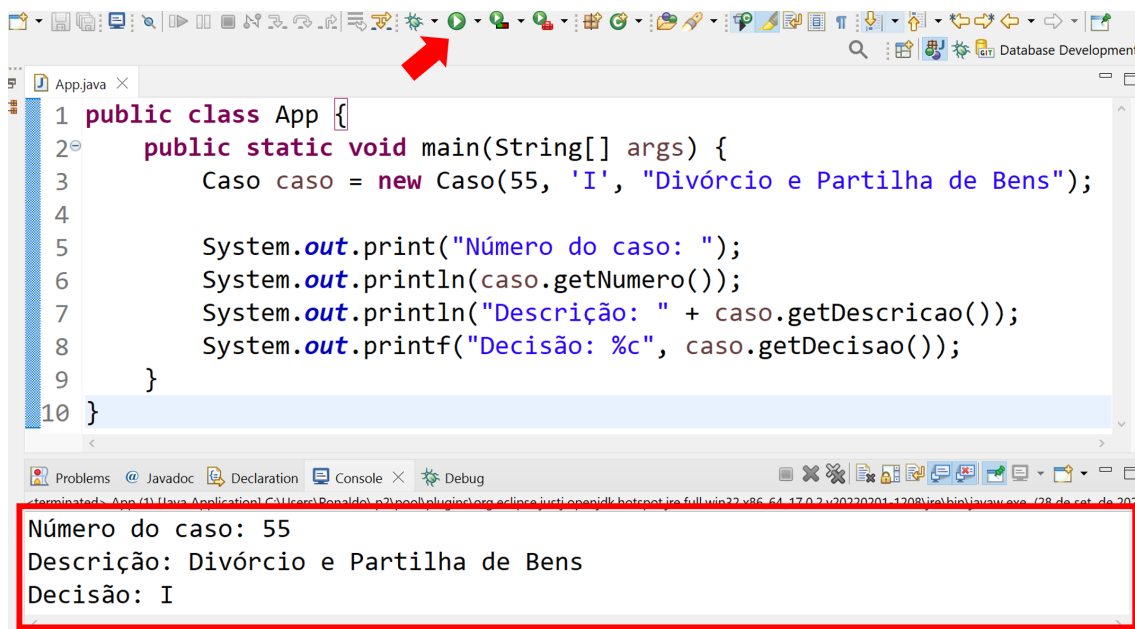
## 5.2 Entrada e saída básica de dados

Uma forma elementar de se comunicar com um programa é através de entrada e saída básica de dados. Ao construir um programa, é possível adicionar instruções que permitem uma interação humano-computador. Dessa forma, podemos atuar como usuários do programa e guiar a execução do código construído. Por vezes, queremos apenas imprimir resultados ou saber o que foi realizado com mais exatidão. Logo, essa tarefa é de grande importância no cotidiano de um programador (SCHILDT, 2015).

Na linguagem Java, realizamos entrada e saída básica de dados para o console por meio da classe *System*. Os atributos públicos *in* e *out* da classe *System* realizam a entrada e a saída de

dados, respectivamente. Como diz o nome, essa classe nos provê uma conexão com o sistema e nos habilita a ler o que está sendo digitado no dispositivo, além de permitir que escrevamos informações na tela do dispositivo.

Vamos começar nossos estudos pela saída de dados. Os métodos mais comuns para imprimir uma saída de texto no console são os métodos: *print*, que apenas imprime o texto passado como parâmetro; *println*, que opera da mesma forma que o *print*, mas pula uma linha após a impressão; e o *printf*, que recebe um parâmetro para imprimir e parâmetros adicionais com valores a serem encaixados no formato desejado.

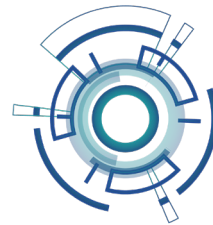


```
1 public class App {
2     public static void main(String[] args) {
3         Caso caso = new Caso(55, 'I', "Divórcio e Partilha de Bens");
4
5         System.out.print("Número do caso: ");
6         System.out.println(caso.getNumero());
7         System.out.println("Descrição: " + caso.getDescricao());
8         System.out.printf("Decisão: %c", caso.getDecisao());
9     }
10 }
```

Número do caso: 55  
Descrição: Divórcio e Partilha de Bens  
Decisão: I

Observe que estamos utilizando agora o Eclipse IDE, para poder executar nosso código e visualizar a saída. Lembrando: você pode executar o código, clicando no botão executar, indicado pela seta vermelha na imagem apresentada. Neste exemplo, criamos um objeto da classe *Caso* e inicializamos seus atributos, com alguns valores. Em seguida, utilizamos o *print*, para escrever a String “Número do caso: ”, e utilizamos o *println*, para escrever o valor do atributo *numero*, na saída do console, e pular uma linha. Subsequentemente, utilizamos o *println* novamente, para imprimir a String “Descrição: ”, aproveitando para concatenar à String o valor do atributo *descricao* e realizar um pulo de linha. Por fim, utilizamos o *printf*, para imprimir “Decisão: I”, pois o parâmetro do tipo *char* terá seu valor encaixado onde temos o símbolo “%c”.

Para realizar entrada de dados, além de utilizar o atributo *in* da classe *System*, iremos utilizar a classe *Scanner*. Esta classe irá nos ajudar a ler a entrada do sistema e capturar os dados digitados, a linha 5 da imagem de exemplo a seguir. Logo depois, digitamos uma mensagem de saída para que o usuário escreva uma descrição para um novo caso a ser registrado. A



execução do programa fica suspensa na linha 8, até que o usuário escreva um texto. O usuário escreve “Testamento e Sucessão” como descrição e aperta a tecla *enter*. Dessa forma, a execução do programa continua na linha 10, onde pedimos para que um número de caso seja informado. Ao digitar “56”, fechamos o *Scanner* através do método *close*, para interromper a leitura do dispositivo. Note que imprimimos os valores, para garantir que a entrada foi lida de forma correta.

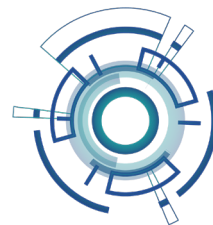
```
1 import java.util.Scanner;
2
3 public class App {
4     public static void main(String[] args) {
5         Scanner scanner = new Scanner(System.in);
6
7         System.out.print("Digite uma descrição do caso: ");
8         String descricao = scanner.nextLine();
9
10        System.out.print("Digite o número do caso: ");
11        int numero = scanner.nextInt();
12
13        scanner.close();
14
15        System.out.println("Caso " + numero + ": " + descricao);
16    }
17 }
```

Problems | Javadoc | Declaration | Console | Debug

<terminated> App (1) [Java Application] C:\Users\Ronaldo\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86\_64\_17.0.2.v20220201-1208\jre\bin\javaw.exe (28 de set. de 2023 19:2

Digite uma descrição do caso: Testamento e Sucessão  
Digite o número do caso: 56  
Caso 56: Testamento e Sucessão

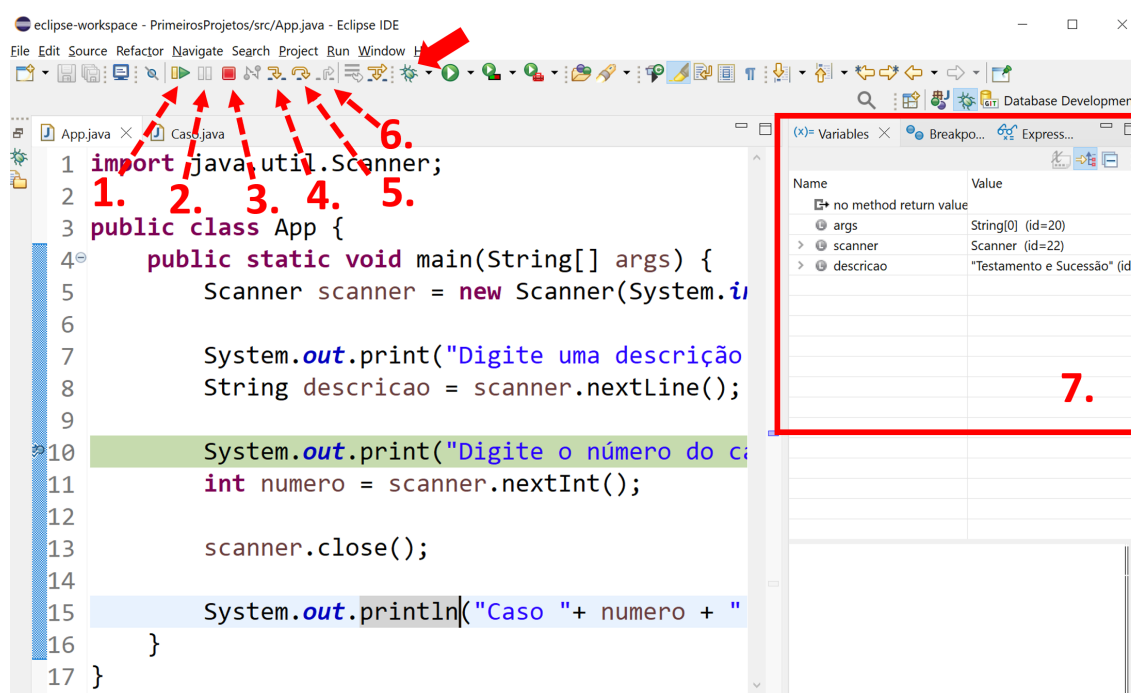
Vários outros métodos podem ser usados para realizar leitura de dados de entrada, como *nextDouble*, *nextBoolean* e assim por diante. Um importante aspecto do uso de entrada e saída para o console é que pode ser utilizado para entender o funcionamento de um programa ou identificar erros no código. Todavia, devemos reconhecer outros utilitários e outras tarefas apropriadas para isso, que estudaremos agora, na próxima seção.



## 6 Depuração de código e enumerações

### 6.1 Depuração de código no Eclipse IDE

A tarefa depuração de código permite que um programador posicione pontos de parada na execução de um programa, *breakpoints*, para que seja possível verificar o estado do programa, em tempo de execução. Um dos principais motivos pelos quais devemos depurar nossos programas é para identificar *bugs* e erros que podem ter sido introduzidos inadvertidamente. Podemos dizer, então, que o processo de depuração faz parte de uma estratégia para testar nosso programa (PRESSMAN, 2021). Vamos seguir nosso exemplo de execução para o Eclipse IDE, conforme mostra a seguinte imagem.



Para executar um programa no Eclipse IDE em modo de depuração ou *debug*, precisamos clicar no ícone de *bug* (em português, “inseto”), conforme mostra a seta cheia, na imagem acima. Ao executar o programa neste modo, caso seja a primeira vez, o Eclipse irá perguntar se você gostaria de mudar a perspectiva para o modo de depuração. Isso nada mais é do que um reposicionamento dos utilitários do ambiente de desenvolvimento, para dar ênfase aos componentes de depuração. Note que, para o exemplo da imagem, estamos utilizando a perspectiva de *debug*.

O programa segue seu fluxo regularmente até que um *breakpoint* seja atingido, como podemos ver na linha 10. Nesse momento, a linha será destacada pela cor verde e podemos controlar o fluxo de execução por meio de algumas opções. Cada opção está destacada, na imagem anterior, com um número e, a seguir, cada número é descrito e explicamos como a respectiva opção funciona:

1. Continuar: esta opção é útil, quando não queremos investigar o fluxo atual e é do nosso interesse liberar a execução normal do programa. Ao clicar no botão continuar, a execução do programa segue o fluxo regular, saindo de um estado de suspensão e continuando a execução até que um *breakpoint* seja atingido ou a execução do programa chegue ao fim. Enquanto a aplicação estiver em execução, este botão estará desabilitado. O botão só poderá ser clicado se a execução estiver suspensa, como é o caso da imagem.
2. Pausar: quando desejar, um programador pode suspender a execução de um programa por meio desta opção, com um símbolo de *pause*. Isto pode ser útil, quando suspeitamos que nosso programa caiu em um *loop* infinito, quando queremos investigar algo que chamou nossa atenção, ou apenas queremos parar a execução. O botão só estará disponível para uso, enquanto o programa estiver em execução. Note, neste exemplo, que, como a execução está suspensa, o botão está desativado.
3. Parar: um programador faz uso da opção parar, quando deseja interromper, imediatamente, o fluxo de execução, dando um fim ao processo de execução atual do programa. Uma vez que esta opção seja usada, o programa sai de um estado de execução e só retorna a executar caso o programador inicie um novo processo de execução pelo botão de *debug*.
4. Entrar: uma vez que a execução esteja suspensa em uma linha, como é o caso deste exemplo, a opção entrar executa o próximo passo do programa, levando em consideração quaisquer chamadas e usos de métodos. Em outras palavras, executamos a próxima instrução. Se for uma chamada de método, por exemplo, o fluxo de execução continua suspenso, mas segue para dentro do método chamado. É possível utilizar essa opção repetidamente, mas isto, em geral, significa investigar as partes mais internas de um programa. Um programador pode fazer uso da opção entrar, quando quer investigar o conteúdo de um método para uma chamada específica ou situação semelhante.
5. Pular: a opção pular executa a próxima instrução do programa. De forma mais precisa, podemos dizer que esta opção executa exatamente uma linha e pula para a próxima linha. Mesmo se houver uma chamada de método na linha executada, diferente do entrar, que investiga a parte de dentro do método chamado, a opção pular apenas executa a chamada, sem entrar no método, e suspende a execução na próxima linha. Esta opção é, possivelmente, a mais utilizada por um programador, pois podemos executar, linha a linha, um programa sem sair do trecho de código sendo investigado.
6. Sair: semelhante à opção entrar, esta opção executa o próximo passo do programa. A principal diferença é que, enquanto no entrar o programador busca investigar detalhes de uma chamada de método, por exemplo, no sair iremos liberar a execução completa de métodos internos, a fim de retornar a suspensão de execução para o ponto original da chamada. Um

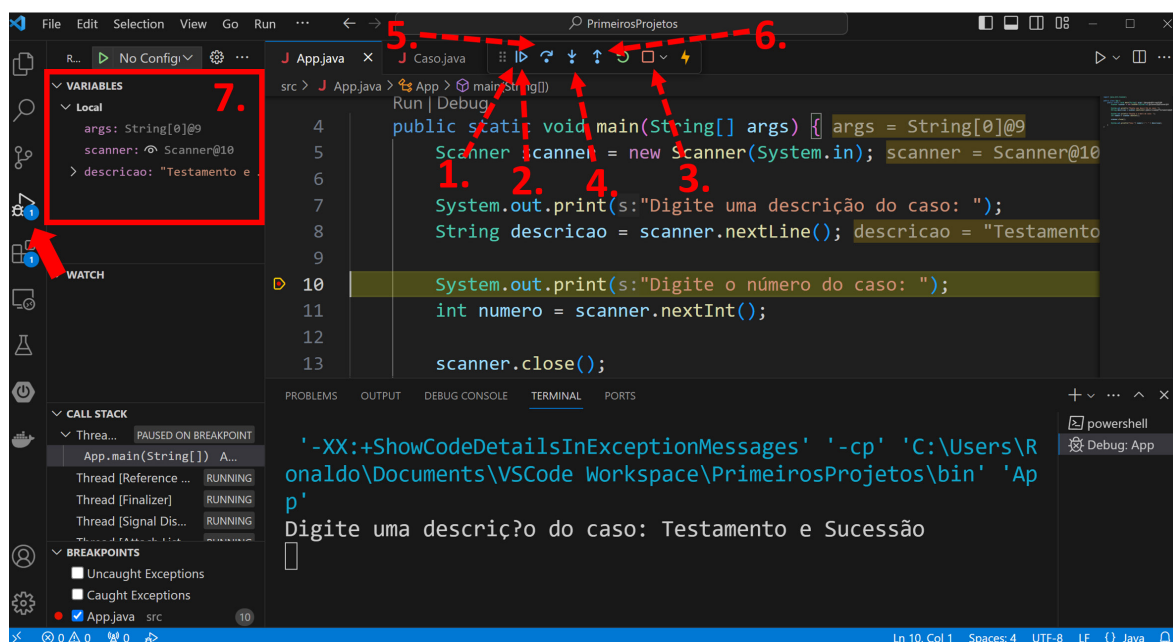
programador pode usar essa opção, após utilizar a opção entrar e ficar satisfeito com suas investigações, desejando retornar ao ponto de execução original.

7. Visualizar detalhes: nesta área, o programador tem a opção de visualizar valores para variáveis, *breakpoints* e demais detalhes de execução do programa. Note que, para a execução do exemplo acima, a variável *descricao* atualmente possui um valor “Testamento e Sucessão”, pois este texto foi digitado no console durante a execução das linhas 7 e 8. Esta área é útil para acompanhar o estado atual dos elementos de execução do programa.

O modo de perspectiva *debug* é uma característica do Eclipse, mas, em geral, todo ambiente integrado de desenvolvimento possui opções semelhantes, para um controle de fluxo de execução em modo de depuração. A seguir, vamos estudar como realizar depuração no VSCode.

## 6.2 Depuração de código no Visual Studio Code

Para executar o mesmo programa em modo depuração no Visual Studio Code, ou VSCode, seguiremos passos muito semelhantes aos efetuados no Eclipse. Vale ressaltar que a maioria das opções são idênticas, mas a ordem das opções, na interface gráfica do ambiente de desenvolvimento, é diferente. A imagem a seguir mostra o modo de depuração no VSCode para o mesmo programa e utilizando a mesma numeração para as opções iguais.



Note que a seta cheia indica o local para depurar o código e os números indicam as exatas mesmas opções encontradas no Eclipse. Uma diferença é que, como a opção continuar (1) só está disponível quando a execução está parada e a opção pausar (2) só está disponível



quando o programa estiver em execução, no VSCode estas opções ocupam o mesmo espaço. Enquanto um botão está disponível, o outro sai da visão do programador. Note que as demais opções são iguais e possuem ícones e áreas semelhantes.

## 6.3 Enumerações

Agora que estudamos classes, objetos e conceitos de programação orientada a objetos básica, incluindo operações de entrada e saída, e manipulação de datas, o último conceito que veremos, nesta trilha de aprendizagem, é o de enumerações. Também conhecidos simplesmente como *enums*, as enumerações em Java são um tipo de dados útil, para representar um conjunto de valores distintos.

Para exemplificar o uso de enumerações, vamos fazer uma última atualização na classe *Caso*, para adicionar um atributo estado, que pode assumir os valores aberto, em andamento, concluído e arquivado. Um *enum* pode ser criado em um arquivo Java próprio, mas, como este estado depende diretamente da classe *Caso*, criaremos esse *enum*, de maneira aninhada na classe.

```
public class Caso {
    private int numero;
    private char decisao;
    private String descricao;
    private Estado estado;

    enum Estado {
        ABERTO, EM_ANDAMENTO, CONCLUIDO, ARQUIVADO
    }

    public Caso(int numero, char decisao, String descricao) {
        this.numero = numero;
        this.decisao = decisao;
        this.descricao = descricao;
        this.estado = Estado.ABERTO;
    }

    // ...
}
```

Nosso *Estado* define quatro valores distintos que podem ser utilizados e criamos também um atributo do estado do tipo *Estado*. Podemos dizer, então, que todo objeto do tipo *Caso*, em tempo de execução, terá um dos quatro valores definidos acima, sendo o primeiro deles, no momento de criação, o estado aberto, conforme escrevemos na última linha do construtor.

Na próxima trilha, estudaremos, com maior profundidade, o conceito de pacotes e veremos como organizar classes que fazem uso de coleções e APIs da linguagem Java. Exploraremos as alternativas utilizadas para relacionar classes distintas e iremos expandir nosso conhecimento de entrada e saída, incorporando operações em arquivos e gerenciamento de memória.





## REFERÊNCIAS

DEITEL, Paul; Deitel, Harvey. **Java: Como Programar**. 10ª edição. Editora Pearson Universidades, 2016.

FERREIRA, Arthur G. **Interface de programação de aplicações (API) e web services**. Editora Saraiva, 2021.

FURGERI, Sérgio. **Java 7 - Ensino Didático**. Editora Saraiva, 2012.

LAMOUNIER, Stella Marys D. **Qualidade de software com Clean Code e técnicas de usabilidade**. Editora Saraiva, 2021.

MACHADO, Rodrigo P.; FRANCO, Márcia H.I.; BERTAGNOLLI, Silvia de C. **Desenvolvimento de software III: programação de sistemas web orientada a objetos em java** (Tekne), 2016.

MARTIN, Robert C. **Desenvolvimento Ágil Limpo**. Alta Books, 2020.

MAYER, Carlos R.; MAZER, Ademir Jr. **Visão Introdutória sobre os conceitos do código limpo**, 2015.

OLIVEIRA, Diego Bittencourt de. *et al.* **Desenvolvimento para dispositivos móveis**. SAGAH, 2019.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de software**, 2021.

ROSEN, Kenneth H. **Matemática Discreta e suas Aplicações**. 8ª edição. McGraw Hill, 2018.

SCHILDT, Herbert. **Java para iniciantes**, 2015.

SEBESTA, Robert. **Conceitos de Linguagens de Programação**, 2018.

SOMMERVILLE, Ian. **Engenharia de software**. Tradução Maurício de Andrade. 9. ed. São Paulo: Pearson Education do Brasil; Addison-Wesley, 2011.