Fillipe Cordeiro

Java Essencial

para Android

AndroidPro

Java Essencial para Android

Fillipe Cordeiro | AndroidPro

1ª edição

Java Essencial para Android

AUTOR Fillipe Cordeiro

SITE www.androidpro.com.br

E-MAIL suporte@androidpro.com.br

Sumário

Introdução

Como sabemos, o Kit de Desenvolvimento de Software da plataforma Android (o Android SDK) é construído utilizando a linguagem Java.

Sendo assim, para todos os desenvolvedores, alunos ou aprendizes que querem se tornar desenvolvedores Android, mas ainda não possuem conhecimentos em Java, este livro servirá como um minicurso composto por três capítulos sobre os fundamentos da linguagem.

Neste ebook, você aprenderá o básico da linguagem de programação Java e Orientação a Objetos, além dos conceitos necessários para o desenvolvimento de aplicativos Android.

Se você já é um programador Java experiente e quer aprender sobre desenvolvimento Android, pode pular a leitura deste livro. Se este não é o caso e você ainda não tem conhecimentos dessa linguagem, é importante ler o livro e praticar os exercícios propostos, para compreender bem a linguagem Java e se tornar apto a avançar para o desenvolvimento Android. Afinal, todos os profissionais de desenvolvimento Android devem compreender completamente o conteúdo deste curso intensivo de Java.

Boa leitura!

Código-fonte dos Exercícios

Todos os exercícios propostos nesse material podem ser acessados pelo endereço do GitHub do AndroidPro.

https://github.com/AndroidProBlog/JavaEssencialAndroid

1. Java Básico

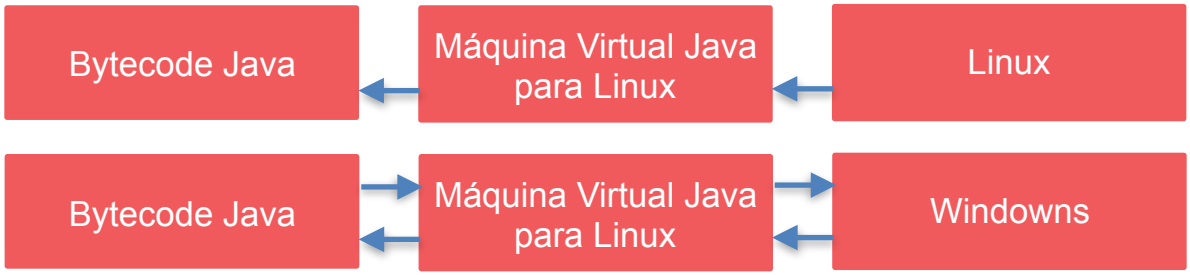
1.1 Linguagem de Programação Java

A linguagem de programação Java foi originalmente desenvolvida pela Sun Microsystems e lançada em 1995. Hoje, é uma das linguagens de programação mais poderosas e populares do mundo.

As aplicações Java são normalmente compiladas para “bytecode”, que pode rodar em qualquer JVM (Java Virtual Machine) independentemente do sistema operacional em execução. O Java é uma linguagem Orientada a Objetos para uso geral, e foi projetada para permitir que os desenvolvedores escrevam o código uma vez e consigam executá-lo em qualquer lugar, ou seja, o código executado em uma plataforma não precisa ser recompilado para executar em outra.

1.2 Java Virtual Machine (JVM)

A Maquina Virtual Java ou Java Virtual Machine (JVM) é o componente do framework Java responsável por executar o código compilado. Quando um desenvolvedor compila um arquivo Java, o compilador Java produz um arquivo de bytecode que tem uma extensão .class. Um bytecode é uma linguagem intermediária produzida por pelo compilador Java, e é executado em uma JVM.



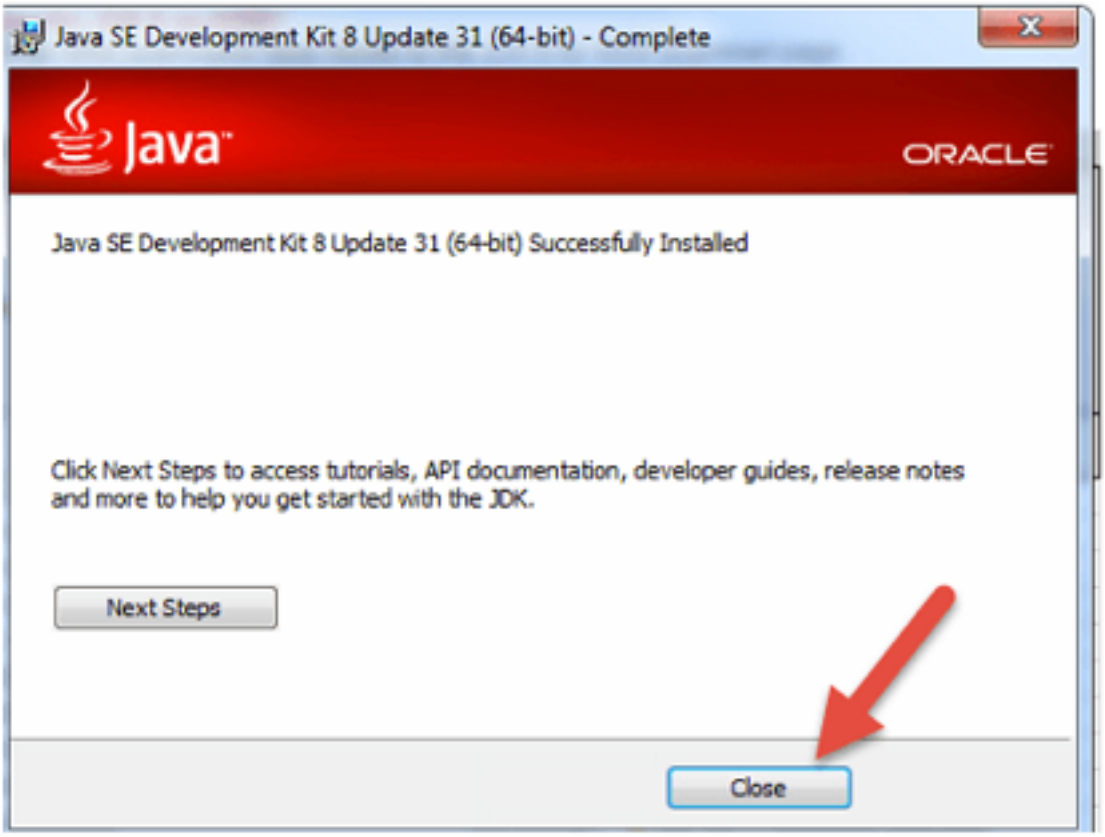
1.3 JDK e JRE

Para começar a programar em Java, o desenvolvedor precisa conhecer e entender alguns componentes principais da linguagem: o Java Development Kit (JDK) e o Java Runtime Environment (JRE). O JDK fornece um compilador Java, além de outras ferramentas que permitem que o programador escreva os códigos Java e possa convertê-los em um arquivo de bytecode para ser executado por uma JVM. O programa que compila o código Java é o javac. O JRE é o ambiente de execução dos programas Java, compilados em um formato binário portátil (arquivos .class) pelo compilador.

Se você instalar o JRE na sua máquina, seu sistema operacional poderá executar programas Java sem nenhum problema, mas para começar a escrever programas em Java você precisa da JDK também.

1.4 Configurando Sua Máquina para Programar em Java

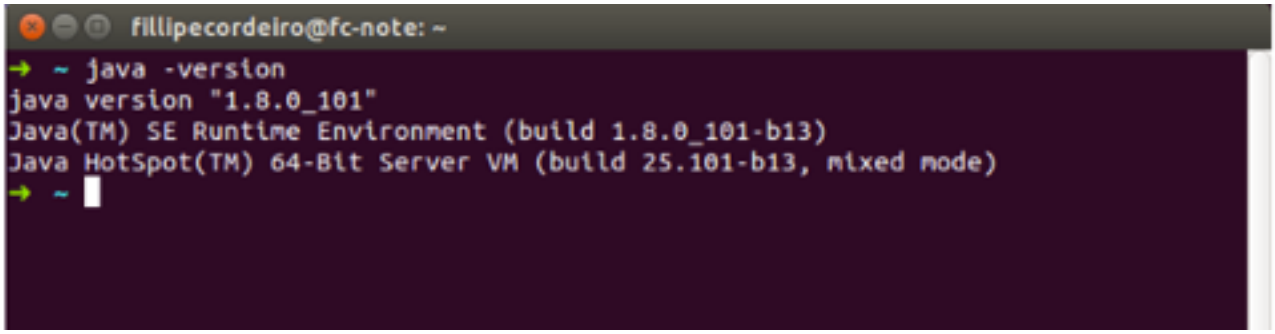
Para começar a programar em Java, você deve instalar o JDK 8 na sua máquina. Você
pode fazer o download na página da Oracle.com - empresa proprietária do Java. Certifique-se de baixar a versão correta, compatível com a sua máquina (32 ou 64 bits). Depois de baixar o instalador e executar o arquivo, você terá a seguinte mensagem:



Então, é só clicar no botão fechar (Close).

Para verificar se você instalou o JDK na sua máquina com sucesso, faça o seguinte:

1. Abra um prompt de comando:
2. Na janela que se abre, digite java -version.
    - No Windows, vá no menu **Iniciar > Executar** e digite **cmd**.
    - No Mac OS ou Linux, abra o Terminal de comandos.
3. Você verá informações semelhantes a estas:



1.5 Olá Mundo em Java

Veremos nesta seção como escrever o famoso programa “Olá Mundo!” em Java usando um editor de texto simples (como o bloco de notas) ou uma IDE, como o Intellij IDEA.

1.5.1 Usando um Editor de Texto

1. Abra um editor de texto simples (como o Bloco de Notas) e digite o seguinte código Java:

```
public class OlaMundo {
    public static void main(String[] args) {
        System.out.println("Olá Mundo!");
    }
}
```

2. Salve o arquivo como **OlaMundo.java** em uma pasta de fácil acesso. Observe que o nome do arquivo é o mesmo que o nome da classe no código.
3. Abra um prompt de comando e compile o código digitando **javac OlaMundo.java**
4. Se não ocorrer nenhum erro, isso significa que a compilação foi bem sucedida e você pode executar o programa. Para executá-lo digite **java OlaMundo**.
5. A linha **"Olá Mundo!"** deve ser impressa.

Depois de compilar um arquivo Java com sucesso, será criado um outro arquivo de extensão .class, este é o arquivo bytecode executado pela JVM.

1.6 Usando uma IDE

Um Ambiente de Desenvolvimento Integrado (IDE) é uma aplicação que fornece um conjunto de ferramentas para ajudar o desenvolvedor a realizar muitas tarefas de forma mais eficiente. Os passos seguintes descrevem como criar, compilar e executar o programa “Olá Mundo” utilizando a IDE Intellij IDEA.

Obs: Estamos utilizando o Intellij IDEA, pois é uma ferramenta bem parecida com o Android Studio para desenvolvimento Android.

1. Crie um novo projeto no Intellij IDEA, que conterá conterá o arquivo Java, chame-o de **JavaEssencialAndroid**. Clique em **File > New > Project**, e escolha **Java**.
2. Clique em **Next** até aparecer o campo para digitar o nome do projeto. Digite o nome e, em seguida, clique em **Finish**.
3. No painel esquerdo, em **Project**, expanda o projeto, em seguida, clique com botão direito do mouse em **src**.
4. Vá ao menu **New > Java Class**.
5. No campo **Name**, digite o nome da classe: **OlaMundo** e clique em **OK**
6. No editor de arquivo, o arquivo OlaMudo.java será aberto. Digite o mesmo código mostrado anteriormente.
7. Clique no botão **Run**.
8. O "Olá mundo!" será exibido na visualização **Console**.

1.7 Tipos de Dados Primitivos do Java

Todas as variáveis em Java devem ser declaradas antes de serem usadas. Isto é feito através da especificação do tipo da variável e do nome da variável:

```
int algumaVariavel = 1;
```

O Java suporta oito tipos de dados primitivos diferentes:

1. **byte**: O tipo de dados byte é um número inteiro com 8 bits.
2. **short**: O tipo de dados short é um inteiro com 16 bits.
3. **int**: O tipo de dados int é um inteiro com 32 bits. Ela tem um valor máximo de 2.147.483.647.
4. **long**: O tipo de dados long é um número inteiro com 64 bits.
5. **float**: O tipo de dados float é um tipo simples de ponto flutuante com 32 bits.
6. **double**: O tipo de dados double é um tipo de ponto flutuante com 64 bits.
7. **boolean**: O tipo de dados boolean tem apenas dois valores possíveis: true e false.
8. **char**: O tipo de dados char é um caractere único, unicode, com 16 bits.

1.8 Arrays

As Arrays são recipientes que armazenam um número fixo de valores de um determinado tipo. O tamanho de uma Array é fixo e é declarado quando a Array é criada.

Para declarar uma Array de dez elementos inteiros:

```
int [] minhaArray = new int [10];
```

Cada item em uma Array é chamado de elemento, e cada elemento é acessado pelo índice numérico. O índice de numeração em Arrays começa com 0. O 10º elemento, por exemplo, é, acessado pelo índice de número 9. Você pode atribuir um valor a um elemento da Array usando a seguinte sintaxe:

Obs.: O programa a seguir, Capitulo1Array, cria uma Array de inteiros, coloca alguns valores, e imprime cada valor para a saída padrão.

```
class Capitulo1Array {
 public static void main(String[] args) {
  // Alocar memória para 5 inteiros
  int[] umArray;
  umArray = new int[5];
  // Inicializa elementos
  umArray[0] = 10;
  umArray[1] = 20;
  umArray[2] = 30;
  umArray[3] = 40;
  umArray[4] = 50;
  System.out.println("Valor no índice 0:" + umArray[0]);
  System.out.println("Valor no índice 1:" + umArray[1]);
  System.out.println("Valor no índice 2:" + umArray[2]);
  System.out.println("Valor no índice 3:" + umArray[3]);
  System.out.println("Valor no índice 4:" + umArray[4]);
 }
}
```

1.9 Controle de Fluxo

Declarações em código Java são executadas sequencialmente, de cima para baixo, na ordem em que elas aparecem. No entanto, um programador pode "controlar o fluxo de execução", utilizando declaração condicional e loops. Esta parte do ebook descreve o uso de declarações de tomada de decisão (if / else e switch), loops (for, while) e as demonstrações de controle de fluxo (break, continue e return).

1.9.1 if/else e switch

As declarações if/else fazem seu programa para executar uma determinada seção de código somente se uma determinada condição for true (verdadeira).

```
if (algumaExpressão) {
codigo1
} else {
codigo2
}
```

Se algumaExpressão for avaliada como **true**, então o codigo1 é executado. Se algumaExpressão for **false**, então o codigo2 é executado.

1.9.2 Switch

Ao contrário das declarações if / else, o **switch** tem vários possíveis caminhos de execução. O **switch** funciona com os tipos de dados primitivos *byte*, *short*, *char*, e *int*.

A seguir veja um exemplo de instrução switch, que imprime saídas com base na variável diaDaSemana:

```
int diaDaSemana = 1;
String diaString = "";
switch (diaDaSemana) {
 case 1:
  diaString = "Segunda-feira";
  break;
 case 2:
  diaString = "Terça-feira";
  break;
 case 3:
  diaString = "Quarta-feira";
  break;
 case 4:
  diaString = "Quinta-feira";
  break;
  case 5: diaString = "Sexta-feira";
  break;
  case 6: diaString = "Sabado";
  break;
  case 7: diaString = "Domingo";
  break;
}
System.out.println(diaString);
```

1.9.3 While

Uma declaração **while** executa continuamente um bloco de código enquanto a condição for **true**.

A sintaxe da linguagem pode ser expressada da seguinte forma:

```
while (expressão) {
    codigo1
}
```

Expressão é uma declaração que deve ser avaliada como um valor booleano (**true** or **false**). Se for **true**, então o bloco a seguir será executado repetidamente, até que a expressão torne-se **false**.

O trecho de código abaixo, irá imprimir o valor da variável **contador** 10 vezes até que se torne
igual a 11:

```
int contador = 1;
while (contador < 11) {
System.out.println("Contador is: " + contador);
contador++;
}
```

1.9.4 For

A declaração **for** proporciona uma forma compacta de iterar sobre um intervalo de valores. Os programadores geralmente se referem a ele como o "for loop" por causa da maneira em que ele repetidamente retorna até que uma determinada condição é satisfeita. A forma geral da declaração **for** pode ser expressa como se segue:

```
for (inicia; condicao; incremento) {
   codigo1
}
```

O código a seguir usa a forma geral da declaração for para imprimir os números de 1 a 10
para a saída padrão:

```
for (int i = 1; i <= 10; i++) {
   System.out.println("Valor de i éis: "+ i);
}
```

Exercício 1

Objetivos:
- Criar um novo pacote Java no projeto JavaEssencialAndroid
- Praticar as declarações de controle em Java

Pré-requisitos:
Crie um novo projeto Java no IntelliJ IDEA antes de resolver esse exercício, seguindo as etapas a seguir:
1. Abra o projeto **JavaEssencialAndroid** no Intellij IDEA
2. Clique com o botão direito do mouse em **src**, vá em **New > Package**, digite o nome **br.com.androidpro.javaandroid.exercicio1**.
3. Clique em **OK**.

Agora você terá um novo pacote dentro do projeto.

Exercício:
Escreva o código Java que imprime os números de 1 a 10, cada um em uma linha separada.

Neste exercício, você vai criar um novo projeto Java no IntelliJ IDEA e escrever o código necessário, em seguida, vai executar o programa.

Crie uma classe Java para escrever seu código
1. Na visualização à esquerda do Intellij IDEA, clique com o botão direito do mouse sobre o seu pacote, criado na pasta **src**
2. No menu que aparece, vá em **New**, em seguida, escolha **Java Class**.
3. No campo **Nome**, digite **Capitulo1**, e clique em **OK**.
4. Dentro da nova classe, digite o código:

```
public static void main (String [] args) {

}
```

Escreva o código Java para resolver o exercício

1. Dentro do método **main()**, digite as seguintes linhas de código:

```
for (int i=1; i<=10; i++) {
    System.out.println(i);
}
```

Execute o programa Java

1. Clique com o botão direito do mouse sobre o nome da sua classe Java.
2. No menu que aparece, vá em **"Run Capitulo1.main()"**.
3. Verifique a exibição do console. Você deve ver os números impressos.

2. Programação Orientada a Objetos

O Java é uma linguagem de programação Orientada a Objetos (OOP), compartilhando os mesmos conceitos e características de outras linguagens OOP. Agora você será apresentado(a) aos objetos, classes, herança e interfaces.

2.1 Objetos

Um objeto é um grupo de estados e comportamentos de software. Objetos no Java são frequentemente utilizados para a representação de objetos que encontramos na vida real, e são essenciais para a compreensão da programação orientada a objetos. Os objetos da vida real dividem duas características: estado e comportamento. Por exemplo, um carro tem um estado (modelo atual, fabricante, cor) e um comportamento (direção, mudança de velocidade etc.)

Construir o seu próprio código utilizando objetos separados proporciona muitos benefícios, incluindo a reutilização de código, ocultar informações, facilidade de depuração e etc.

2.2 Classes

Uma classe é um protótipo a partir do qual os objetos são criados. As Classes fornecem uma maneira limpa para modelar o estado e o comportamento de objetos do mundo real.
As duas propriedades principais que definem uma classe são: um conjunto de **variáveis de classe** (também chamadas de **campos/atributos**), e um conjunto de **métodos** de classe (ou **funções**).

Os métodos e as funções significam a mesma coisa no contexto da programação orientada a objetos e ambos são utilizados alternadamente nesse ebook.

Para representar o estado de um objeto nas classes, adicione atributos a uma classe. Os comportamentos dos objetos são representados usando métodos. A seguir, segue uma classe Java simples chamada Veículo.

```
class Veiculo {

 int velocidade = 0;
 int marcha = 1;

 void trocarMarcha(int novaMarcha) {
  marcha = novaMarcha;
 }

 void acelerar(int incremento) {
  velocidade = velocidade + incremento;
 }

 void mostrar() {
  System.out.println("Velocidade:" + velocidade + " Marcha:" + marcha);
 }

}
```

O estado do objeto Veículo é representado com os atributos velocidade e marcha. O comportamento do objeto pode ser alterado por meio de dois métodos: **trocarMarcha()** e **acelerar()**.

2.3 Getters e Setters

Geralmente cria-se um conjunto de métodos em uma classe, para ler/escrever especificamente os valores dos atributos de classe. Estes são chamados **getters** (usados para obter os valores) e **setters** (usados para alterar os valores). Os **Getters** e **Setters** são cruciais para as classes Java, uma vez que são usados para gerenciar o estado de um objeto.

Na classe de Veículo fornecida anteriormente, podemos adicionar dois métodos (um getter e um setter) para cada atributo. A seguir está o código completo da classe depois de adicionar os getters e setters:

```
class Veiculo {

 int velocidade = 0;
 int marcha = 1;

 public int getVelocidade() {
  return velocidade;
 }

 public void setVelocidade(int v) {
  velocidade = v;
 }

 public int getMarcha() {
  return marcha;
 }

 public void setMarcha(int m) {
  marcha = m;
 }

 void trocarMarcha(int marcha) {
  marcha = novaMarcha;
 }

 void acelerar(int incremento) {
  velocidade = velocidade + incremento;
 }

 void mostrar() {
  System.out.println("Velocidade:" + velocidade + " Marcha:" + marcha);
 }
}
```

Você pode usar o IntelliJ IDEA para gerar os getters e setters para você. Clique no botão direito do mouse dentro do código da classe e escolha **Generate**, em seguida, **Getters and Setters**, e depois selecione os atributos.

2.4 Herança

A Herança fornece um mecanismo poderoso e natural para organizar e estruturar o seu software. Ela estabelece uma relação de Pai-Filho entre dois objetos diferentes.

A programação orientada a objetos permite que classes herdem estado e comportamento utilizados a partir de outras classes. No exemplo abaixo, o Veículo torna-se o pai (superclasse), do Caminhão e do Carro. Dentro da programação em Java, cada classe pode ter uma superclasse direta, e cada superclasse tem o potencial para um número ilimitado de subclasses.

```
public class Carro extends Veiculo {
 int numeroLugares;
}

public class Caminhao extends Veiculo {
 int capacidadeCarga;
}
```

O Carro e o Caminhão agora compartilham o mesmo estado e comportamento definido na classe Veículo.

2.5 Palavras-chave this e super

Existem duas palavras-chave no Java que você vai encontrar quando utilizar Herança nas classes: **this** e **super**. A palavra-chave **this** é usada como uma referência para a própria classe pai, enquanto **super** é uma referência para a classe pai de onde a classe this veio. Em outras palavras, super é usada para o acesso aos atributos e métodos da classe pai.

A palavra-chave **super** é especialmente útil quando você quer substituir o método de uma superclasse em uma classe filha, mas quer chamar o método da superclasse. Por exemplo, na classe Carro, você pode substituir o método **mostrar()** e chamar o **mostrar()** do Veículo:

```
public class Carro extends Veiculo {
 int numeroLugares;
 void mostrar() {
  super.mostrar();
  System.out.println(" Numero de Lugares:" + numeroLugares);
 }
}
```

Chamar o método mostrar() da classe Carro, irá chamar primeiro o método mostrar() da classe Veiculo e depois imprimir o numero de lugares.

2.6 Interface

Uma interface é um contrato entre uma classe e o mundo exterior. Quando uma classe implementa uma interface, ela deve fornecer o comportamento especificado por essa interface.
Vamos usar o exemplo dos veículos, e criar uma interface para ele.

```
public interface IVeiculo {
 void trocarMarcha(int novaMarcha);
 void acelerar(int incremento);
}
```

Em seguida, a classe do Veículo implementa a interface IVeiculo usando a seguinte sintaxe:

```
class Veiculo implements IVeiculo {
 int velocidade = 0;
 int marcha = 1;

 public void trocarMarcha(int novaMarcha) {
  marcha = novaMarcha;
 }

 public void acelerar(int incremento) {
  velocidade = velocidade + incremento;
 }
```

```
void mostrar() {
  System.out.println("Velocidade:" + velocidade + " Marcha:" + marcha);
 }
}
```

Observe que a classe Carro deve fornecer e implementar os métodos **trocarMarcha()** e **acelerar()**.

2.7 Modificadores de Acesso

Os modificadores de acesso determinam se outras classes podem usar um atributo em particular ou invocar um determinado método. Existem quatro tipos de controle de acesso:

- Na classe: public, ou por padrão (sem modificador explícito).
- No método/atributo: public, private, protected ou padrão (sem modificador explícito).

Uma classe pode ser declarada com o modificador **public**, caso em que essa classe é visível para todas as classes, em toda parte. Se uma classe não tem nenhum modificador (o padrão), ela é visível somente dentro de seu próprio pacote (pacotes são grupos de classes relacionadas por nome).

No método/atributo, além do modificador **public** ou sem modificador (pacote-privado), há dois modificadores de acesso adicionais: **private** e **protected**. O modificador **private** especifica que o método/atributo só pode ser acessado em sua própria classe. O modificador **protected** especifica que o método/atributo só pode ser acessado no seu próprio pacote e, em adição, por qualquer outra subclasse da classe.

Modificador	Classe	Pacote	Subclasse	Globalmente
Public	Sim	Sim	Sim	Sim
Protected	Sim	Sim	Sim	Não
Sem Modificador (Padrão)	Sim	Sim	Não	Não
Private	Sim	Não	Não	Não

2.8 Construtores

São invocados para criar objetos. Eles são semelhantes às funções, mas diferenciadas no seguinte:

- Construtores têm o mesmo nome que a classe.
- Eles não têm qualquer tipo de retorno.

Chamar um construtor para criar um novo objeto serve para inicializar os atributos de um objeto. Vamos supor que o Veículo tenha o seguinte construtor:

```
public Veiculo(int v, int m) {
 velocidade = v;
 marcha = m;
}
```

Para criar um novo objeto do tipo Veículo seria necessário invocar o construtor com a palavra-chave **new**:

```
Veiculo veiculo = new Veiculo(100, 5);
```

Esta linha vai criar um objeto do tipo Veículo, e tem os seus dois atributos **velocidade** e **marcha** inicializados a 100 e 5 consecutivamente.

2.9 Overriding e Overloading de Métodos

Dentro da mesma classe, você pode criar dois métodos com o mesmo nome, mas diferentes no número de argumentos e tipos. Isto é chamado de sobrecarga de método (overloading). Observe que alterar o tipo de retorno por si só não é permitido para sobrecarregar um método. Você deve alterar os parâmetros de assinatura, se necessário.

O sobrescrever um método (overriding), ocorre quando uma classe herda um método de uma classe super, mas fornece sua própria implementação desse método. No código a seguir, a classe Carro substitui o método acelerar() definido na classe Veículo.

```
public class Carro extends Veiculo {
 int numeroLugares;
 public void acelerar(int incremento) {
  acelerar = acelerar + incremento + 2;
 }
}
```

Suponha que você crie um objeto do tipo de carro e chame o método de acelerar(). O método do veículo é ignorado e será executado o método que está dentro da classe Carro:

```
Carro carro = new Carro();
carro.acelerar(100);
```

2.10 Polimorfismo

No contexto da programação orientada a objetos, polimorfismo significa que diferentes subclasses da mesma classe pai podem ter diferentes comportamentos, e, ainda assim, compartilhar de algumas das funcionalidades da classe pai.

Para demonstrar o polimorfismo, vamos utilizar o método mostrar() da classe Veículo. Esse método imprime todas as informações em um objeto do tipo Veículo:

```
public void mostrar() {
 System.out.println("Veiculo: Velocidade " + this.velocidade + " Marcha " + this.marcha);
}
```

No entanto, se a subclasse Caminhão utilizar este método, o atributo **capacidadeCarga** não será impressa, porque ela não é um atributo da classe pai Veículo. Para resolvemos isso, podemos substituir o método **mostrar()** como se segue:

```
public void mostrar() {
 super.mostrar();
 System.out.println("Caminhão: Capacidade de Carga " + this.capacidadeCarga);
}
```

Observe que o método **mostrar()** de Caminhão, vai chamar **mostrar()** do pai e adicionar a ele o seu próprio comportamento, que imprime o valor de **capacidadeCarga**.

Nós podemos fazer a mesma coisa com a classe carro.

```
public void mostrar() {
super.mostrar();
System.out.println("Carro: Lugares" + this.numeroLugares);
}
```

Agora, para testar o comportamento polimórfico, vamos criar 3 objetos, cada um de diferentes tipos de Veículos:

```
class Polimorfismo {
 public static void main(String[] args) {
  Veiculo veiculo1, veiculo2, veiculo3;
  veiculo1 = new Veiculo(50, 2);
  veiculo2 = new Carro(50, 2, 4);
  veiculo3 = new Caminhao(40, 2, 500);
  System.out.println("Veiculo 1 info:");
  veiculo1.mostrar();
  System.out.println("Veiculo 2 info:");
  veiculo2.mostrar();
  System.out.println("Veiculo 3 info:");
  veiculo3.mostrar();
 }
}
```

Uma vez que executar essa classe, teremos três saídas diferentes.

```
Veiculo 1 info:
Veiculo: Velocidade 50 Marcha 2
Veiculo 2 info:
Veiculo: Velocidade 50 Marcha 2
Carro: Lugares 4
Veiculo 3 info:
Veiculo: Velocidade 40 Marcha 2
Caminhao: Capacidade 500
```

No exemplo acima, a JVM chamou o método de cada objeto, em vez de chamar o método da classe Veículo para os três objetos.

Exercício 2

Objetivos:
- Implementar relação pai-filho em Java
- Utilizar modificadores de acesso
- Compreender a sobrescrita de método
- Compreender a sobrecarga de método

Pré-requisitos:
Crie um novo pacote Java no projeto **JavaEssencialAndroid**, antes de resolver esses exercícios (de acordo com os passos do Exercício 1).

Exercício:
Crie três classes e uma interface: a classe Veiculo, que implementa a interface IVeiculo, e as classes filhas Carro e Caminhao que são uma extensão da classe Veiculo. Substitua um método de Veiculo e sobrecarregue outro na filha.

Crie um pacote Java
1. Na visualização **Project** à esquerda no Intellij IDEA, clique com o botão direito do mouse sobre a pasta **src**.
2. No menu que aparece, vá em **New > Package**.
3. Na caixa de diálogo **"New Package"**, digite no campo o seguinte: **br.com.androidpro.javaandroid.exercicio2**
4. Clique em **OK**

Crie três classes Java para escrever seu código
1. Na visualização **Project** à esquerda no Intellij IDEA, clique com o botão direito do mouse sobre o novo pacote.
2. No menu que aparece, vá em **New > Java Class**, e digite **Veiculo** no nome.
3. Clique em **OK**
4. Faça a mesma coisa para as classes **Carro** e **Caminhao**

Crie uma interface Java

1. Na visualização **Project** à esquerda no Intellij IDEA, clique com o botão direito do mouse sobre o pacote.
2. No menu que aparece, vá em **New > Java Class**, selecione a opção Interface em **Kind**
3. Digite o nome **IVeiculo** e clique em **OK**

Escreva o código Java para resolver o exercício

1. Abra o arquivo **IVeiculo.java**
2. Digite nele o seguinte código

```
public interface IVeiculo {
    void acelerar(int incremento);

    void trocarMacha(int novaMarcha);
}
```

3. Abra o arquivo Veiculo.java
4. Digite o seguinte código

```
public class Veiculo implements IVeiculo {

    private int velocidade;

    private int marcha;

   public Veiculo(int velocidade, int marcha) {
        this.velocidade = velocidade;
        this.marcha = marcha;
    }

    @Override
    public void acelerar(int incremento) {
        this.velocidade = this.velocidade + incremento;
    }

    @Override
    public void trocarMacha(int novaMarcha) {
        this.marcha = novaMarcha;
    }

    public void mostrar() {
        System.out.println("Veiculo: Velocidade " + this.velocidade + " Marcha " + this.marcha);
    }
}
```

5. Abra o arquivo **Carro.java**
6. Digite o seguinte código

```
public class Carro extends Veiculo {

    private int numeroLugares;

    public Carro(int velocidade, int marcha, int numeroLugares) {
        super(velocidade, marcha);
        this.numeroLugares = numeroLugares;
    }

    @Override
    public void mostrar() {
        super.mostrar();
        System.out.println("Carro: Lugares " + this.numeroLugares);
    }
}
```

7. Abra o arquivo Caminhao.java
8. Digite o seguinte código

```
public class Caminhao extends Veiculo {

    private int capacidadeCarga;

    public Caminhao(int velocidade, int marcha, int capacidadeCarga) {
        super(velocidade, marcha);
        this.capacidadeCarga = capacidadeCarga;
    }

    @Override
    public void mostrar() {
        super.mostrar();
        System.out.println("Caminhao: Capacidade " + this.capacidadeCarga);
    }
}
```

3. Mais Programação em Java

Depois de aprender os princípios básicos de programação orientada a objetos no capítulo anterior, vamos ver alguns tópicos adicionais, que você provavelmente irá ver durante a codificação Java.

3.1 Exceções

Lidar com erros é uma parte essencial quando se escreve um código robusto. O Java usa exceções para lidar com os erros.

Quando ocorre um erro, o ambiente de execução Java lida com um objeto de erro que é criado pelo método onde o erro ocorre. Este objeto é chamado exceção, e contém informações básicas sobre o erro (como o tipo de erro, local, pilha de métodos que levaram para o erro, etc). O processo de criação e manipulação de um objeto de exceção pelo sistema é chamado de lançar uma exceção.

A lista de métodos que levam ao erro é chamado de **Stack**. Ao manipular o erro, o sistema procura por essa pilha, para encontrar que gerou o erro no código. Todos os objetos de exceção são filhos da classe **pai Exception**. Os diferentes tipos ou erros jogados são filhos da classe **Exception**.

Exceções em Java podem ser classificadas em dois tipos:

3.1.1 Exceção Verificada (Checked Exception)

São erros dentro da aplicação, e espera-se que um programador que pretende criar um código robusto bem escrito, se trate esses erros. Por exemplo, a leitura a partir de um arquivo no disco. Neste caso, o programador deve tratar um **java.io.FileNotFoundException** e, posteriormente, notificar o usuário de forma adequada.

3.1.2 Exceção Não Verificada (Unchecked Exception)

Existem dois tipos: **Errors** e **Runtime**. Elas estão agrupadas na mesma categoria porque ambas não podem ser previstas, ou tratadas por um programador.

Errors são externos aos aplicativos. Por exemplo, suponha que um aplicativo abra um arquivo, mas é incapaz de ler o arquivo por causa de um hardware ou sistema defeituoso. A leitura mal sucedida vai lançar **java.io.IOError**, imprimir uma de pilha de erros e sair.

Exceções **Runtime** geralmente indicam bugs de programação, tais como erros lógicos. Eles são da classe **RuntimeException**.

A manipulação de erros (exceções) em Java é feita através dos blocos **try-catch-finally**. Enquanto o bloco **finally** é opcional, o **try** e **catch** são obrigatórios para fazer o tratamento de erros.

Vejamos o seguinte código:

```
public class Capitulo3Excecoes {
    public static void main(String[] args) {

        System.out.println("Ola Mundo, AndroidPro!");

        String stringNula = null;

        System.out.println("Tentando executar...");

        String stringParcial = stringNula.substring(1);

        // A execução será parada antes dessa linha
        System.out.println("String parcial: " + stringParcial);
    }
}
```

Executar o código acima vai resultar em um erro jogado do tipo **NullPointerException**, especificamente na linha **“String stringParcial = stringNula.substring(1);”**, onde estamos tentando ler a partir de um objeto String que é nulo (não inicializado). Para tratar adequadamente esse erro, devemos modificar o código acima para:

```
public class Capitulo3Excecoes {
    public static void main(String[] args) {

        System.out.println("Ola Mundo, AndroidPro!");

        String stringNula = null;

        try {
            System.out.println("Tentando executar...");

            String stringParcial = stringNula.substring(1);

            // A execução será parada antes dessa linha
            System.out.println("String parcial: " + stringParcial);
        } catch (Exception e) {
            System.out.println("Ocorreu um erro: "+ e.getMessage());
        }
    }
}
```

Em vez de quebrar a execução do código e interromper o programa, este código irá tratar o
NullPointerException, imprimindo os detalhes do erro e continuando a execução após o bloco **catch**.

O bloco **finally** pode ser usado após o bloco de **catch**. Este bloco de código sempre será executado, havendo uma exceção lançada ou não.

```
public class Capitulo3Excecoes {
    public static void main(String[] args) {

        System.out.println("Ola Mundo, AndroidPro!");

        String stringNula = null;

        try {
            System.out.println("Tentando executar...");

            String stringParcial = stringNula.substring(1);

            // A execução será parada antes dessa linha
            System.out.println("String parcial: " + stringParcial);
        } catch (Exception e) {
            System.out.println("Ocorreu um erro: "+ e.getMessage());
        } finally {
            System.out.println("Essa linha sempre será executada");
        }
    }
}
```

Usamos o bloco ***finally*** para bloquear, em muitos casos, onde há alguns recursos que precisam ser libertados, mas uma exceção pode nos impedir de fazê-lo. Por exemplo, quando estiver lendo a partir de um arquivo, um programa bem escrito deve fechar o arquivo depois de terminada a leitura e/ou escrita nele. Se uma exceção foi lançada, a linha de código que fecha o arquivo pode ser ignorada. O bloco ***finally*** seria o melhor lugar para se fechar o arquivo.

3.2 Java Collections

O Java fornece um conjunto de classes e interfaces para ajudar os desenvolvedores a lidar com coleções de objetos. Estas classes de coleção são semelhantes a uma Array, exceto pelo seu tamanho, que pode crescer de forma dinâmica durante o tempo de execução. A seguir, você terá uma visão geral de algumas das classes mais populares do **Java Collections**.

3.2.1 Interfaces

As **Java Collections** estão localizadas principalmente no pacote **java.util**. Ele fornece duas interfaces principais: **Collection** e **Map**. Estes dois são o núcleo do framework Java Collection. Existem outras interfaces que são filhas desses dois tipos. Por exemplo, as interfaces **List** e **Set** vêm a partir da interface **Collection**.

Todas estas interfaces são genéricas; ou seja, o tipo de objeto contido na coleção deve ser
especificado pelo programador. Há uma grande diferença entre subclasses da interface **Collection** e as da interface **Map**.

A **Collection** contém um grupo de objetos que podem ser manipulados e repassados. Os elementos podem ser duplicados ou únicos, dependendo do tipo de subclasse. Um **Set**, por exemplo contém apenas objetos únicos. A interface **Map**, no entanto, mapeia chaves para valores e não pode conter chaves duplicadas. Cada chave só pode mapear um valor, no máximo.

3.2.2 Implementações

As seguintes classes são implementações das interfaces citadas no tópico anterior.

ArrayList
Uma **ArrayList** é uma implementação redimensionável da interface List. Ele implementa todos tipos de operações de lista, e aceitam todo tipo de elemento, incluindo nulo. Também fornece métodos para manipular o tamanho da array que é usado internamente para armazenar a lista.

```
import java.util.*;

public class TestaArrayList {

    public static void main(String[] args) {
        // Criar uma lista
        ArrayList<String> androidList = new ArrayList<String>();
        // Adicionando elementos
        androidList.add("Donut");
        androidList.add("Eclair");
        androidList.add("Froyo");
        androidList.add("Gingerbread");
        androidList.add("Honeycomb");
        androidList.add("Ice Cream Sandwich");
        androidList.add("Jelly Bean");
        System.out.println("Tamanho da ArrayList:" + androidList.size());
        // Mostra o conteúdo da lista
        System.out.println("O ArrayList tem os seguintes elementos:" + androidList);
        // Remove elementos da lista
        System.out.println("Excluindo o segundo elemento ...");
        androidList.remove(3);
        System.out.println("Tamanho após eliminações:" + androidList.size());
        System.out.println("Conteúdo após eliminações:" + androidList);
    }
}
```

Segue a saída do programa:

```
Tamanho da ArrayList:7
O ArrayList tem os seguintes elementos:[Donut, Eclair, Froyo, Gingerbread, Honeycomb,
Ice Cream Sandwich, Jelly Bean]
Excluindo o segundo elemento ...
Tamanho após eliminações:6
Conteúdo após eliminações:[Donut, Eclair, Froyo, Honeycomb, Ice Cream Sandwich,
Jelly Bean]
```

HashSet

Essa classe implementa a interface **Set** e permite elementos nulos. Esta coleção não permite elementos duplicados, e cria uma coleção que utiliza uma tabela **hash** para armazenamento. A tabela **hash** armazena informações usando um mecanismo chamado de **hashing**, onde o valor armazenado é usado para determinar uma chave única, que é utilizada como índice no qual os dados são armazenados. A vantagem do **hashing** é que ele permite uma execução rápida para operações básicas, como **add()** e **remove()**.

```
import java.util.HashSet;

public class TestaHashSet {
    public static void main(String[] args) {
        // Criação de um HashSet
        HashSet<String> androidHash = new HashSet<String>();
        // Adicionando elementos
        androidHash.add("Eclair");
        androidHash.add("Eclair");
        androidHash.add("Gingerbread");
        androidHash.add("Gingerbread");
        androidHash.add("Honeycomb");
        androidHash.add("Ice Cream Sandwich");
        androidHash.add("Jelly Bean");
        androidHash.add("Jelly Bean");

        System.out.println("O conteúdo do HashSet:" + androidHash);
    }
}
```

A saída do programa é:

```
O conteúdo do HashSet:[Gingerbread, Jelly Bean, Eclair, Ice Cream Sandwich,
Honeycomb]
```

Observe que há um elemento "Gingerbread" e um elemento "Eclair" no **HashSet**, embora cada um tenha sido adicionado duas vezes no código.

HashMap

É uma implementação baseada em tabela **hash** da interface **Map**. Ela também permite elementos nulos.

O programa a seguir mapeia nomes para os saldos das contas, trata-se de uma ilustração da **HashMap**.

```
import java.util.*;

public class TestaHashMap {
    public static void main(String[] args) {
        // Criação de um HashMap
        HashMap<String, Double> androidMap = new HashMap<String, Double>();
        // Adicionando elementos
        androidMap.put("Donut", new Double(1.6));
```

```
        androidMap.put("Eclair", new Double(2.1));
        androidMap.put("Froyo", new Double(2.2));
        androidMap.put("Gingerbread", new Double(2.3));
        androidMap.put("Honeycomb", new Double(3.1));
        androidMap.put("Ice Cream Sandwich", new Double(4.0));
        androidMap.put("Jelly Bean", new Double(4.1));
        // Obter um conjunto de entradas
        Set<Map.Entry<String, Double>> set = androidMap.entrySet();
        // Obter um iterador
        Iterator<Map.Entry<String, Double>> i = set.iterator();
        // Elementos de indicação
        while (i.hasNext()) {
           Map.Entry<String, Double> me = (Map.Entry<String, Double>) i.next();
           System.out.println(me.getKey() + ":" + me.getValue());
        }
        System.out.println();
        // Aumentar o número da versão do Eclair
        Double versao = androidMap.get("Eclair");
        androidMap.put("Eclair", new Double(versao + 0.1));
        System.out.println("Novo número da versão do Eclair:" + androidMap.get("Eclair"));
    }
}
```

A saída do programa é:

```
Donut:1.6
Gingerbread:2.3
Jelly Bean:4.1
Froyo:2.2
Eclair:2.1
Ice Cream Sandwich:4.0
Honeycomb:3.1

Novo número da versão do Eclair:2.2
```

Exercício 3

Objetivos:
- Escrever um código que utiliza os métodos do ArrayList.
- Aprender os benefícios do Set.

Pré-requisitos:
Crie um novo pacote Java no projeto **JavaEssencialAndroid** chamado **br.com.androidpro.javaandroid.exercicio3**, antes de resolver esses exercícios.

Exercício:
Escreva um programa que adiciona dez Strings em um ArrayList. As Strings devem ter o seguinte formato:

"Elemento - X", onde X é um número entre 1 e 10 e demonstrar a utilização de métodos: **add()**, **remove()** e **indexOf()**.

Crie um pacote Java
1. Na visualização **Project** à esquerda no Intellij IDEA, clique com o botão direito do mouse sobre a pasta **src**.
2. No menu que aparece, vá em **New > Package**.
3. Na caixa de diálogo "New Package", digite no campo o seguinte: **br.com.androidpro.javaandroid.exercicio3**
4. Clique em **OK**

Crie uma classe Java para escrever seu código
1. Na visualização **Project** à esquerda no Intellij IDEA, clique com o botão direito do mouse sobre o novo pacote.
2. No menu que aparece, vá em **New > Java Class**, e digite **Capitulo3** no nome.
3. Clique em **OK**

4. Dentro da nova classe, digite o código:

```
public static void main (String [] args) {

}
```

Escreva o código Java para resolver o exercício

1. Abra o arquivo Capitulo3.java
2. Dentro de método **main()** digite o seguinte código:

```
ArrayList<String> arrayList = new ArrayList <String> ();
 for ( int i = 1; i <= 10; i ++) {
    arrayList.add ( "Elemento - " + i);
 }
 System.out.println("Índice do Elemento 7: " + arrayList.indexOf ("Elemento - 7"));
 arrayList.remove (4);
 System.out.println("Índice do Elemento 7: " + arrayList.indexOf ("Elemento - 7"));
```

Execute o programa Java

1. Clique com o botão direito do mouse sobre o nome da sua classe Java.
2. No menu que aparece, vá em **"Run Capitulo3.main()"**.
3. Verifique a exibição do console. Você deve ver os números impressos.

Verifique a exibição do console

Você deve ver a saída desejada.

```
Índice do Elemento 7: 6
Índice do Elemento 7: 5
```

4. Java Intermediário

Nesse capítulo, você vai mergulhar em um outro conjunto de recursos de Orientação a Objetos, e tópicos mais avançados do Java.

4.1 Classes aninhadas

Usando o Java, você pode definir uma classe dentro de outra classe. Estas são chamadas de classes aninhadas:

```
public class ClassePrincipal {
...
class ClasseAninhada {
...
}
```

As classes aninhadas podem ser estáticas.

```
public class ClassePrincipal {
...
public static class ClasseAninhadaEstatica {
...
}
class ClasseAninhada {
...
}
```

Uma classe aninhada faz parte da classe principal. As classes internas não-estáticas têm acesso a atributos da classe principal, mesmo que sejam declarados privados. No entanto, as classes internas estáticas não tem acesso. Elas são semelhantes a variáveis e métodos de classe, uma classe interna pode ser declarada como **private**, **public** ou **protected**.

4.2 Benefícios de Classes Internas

A seguir estão algumas razões para um programador usar as classes internas:
Melhorar o agrupamento lógico das classes, que são usadas em apenas um lugar. Se uma classe B é útil apenas em uma Classe A, então é lógico fazer da classe B uma classe interna da classe A.
Aumento de encapsulamento. Se a classe B precisa acessar métodos e atributos privados da classe A, o programador pode esconder a classe B dentro de A e manter todos os métodos e atributos privados, e, ao mesmo tempo esconder a classe B de classes externas.
Melhorar a legibilidade do código e facilidade de manutenção. Criando classes internas dentro de uma classe externa fornece uma forma mais clara do código.

4.3 Atributos de Classe Estáticos

Quando criamos vários objetos da mesma classe, cada objeto (instância) tem sua própria cópia distinta de atributos de classe. Às vezes, podemos querer um atributo que é comum a todos os objetos da mesma classe. Para conseguir isso, usamos o modificador static.

Atributos de classe que têm o modificador static na sua declaração são chamados campos estáticos. Eles estão associados à classe, mais do que com qualquer objeto. Cada instância da classe compartilha o mesmo atributo estático, que é salvo em um local fixo da memória.

Qualquer objeto pode alterar o valor de um atributo estático, mas atributos estáticos também podem ser manipulados sem a criação de uma instância da classe.

Por exemplo, vamos modificar a classe Carro do capitulo anterior, adicionando um atributo estático. O atributo **numeroLugares** pode ter valores diferentes para diferentes objetos do tipo de Carro. No entanto, podemos adicionar o atributo estático chamado **numeroCarros** que serão utilizados para acompanhar o número de objetos criados.

```
public class Carro extends Veiculo {

    private int numeroLugares;

    public static int numeroCarros;

    ...
}
}
```

Atributos estáticos são referenciados pelo próprio nome da classe, para deixar claro que eles são podem ser acessados sem precisar de uma instância, como:

```
Carro.numeroCarros
```

Você pode chamar atributos estáticos por uma variável de objeto:

```
Carro carro = new Carro(40, 2, 4);
carro.numeroCarros;
```

Mas isso não é recomendado, uma vez que o atributo estático ficará parecido com um atributo do objeto.

4.4 Métodos Estáticos

O Java também suporta métodos estáticos, bem como atributos estáticos. Os métodos estáticos que têm o modificador **static** em sua assinatura, deve ser chamado pela classe, sem a necessidade da criação de uma instância da classe, como em **Classe.metodo(args)**.

Você também pode chamar métodos estáticos com uma referência de objeto:

```
Carro carro = new Carro(40, 2, 4);
carro.getNumeroCarros();
```

Mas isso não é recomendado, uma vez que o atributo estático ficará parecido com um atributo do objeto.

Um uso comum para métodos estáticos é acessar campos estáticos. Por exemplo, vamos modificar a classe Carro pela adição de um método estático que retorna os **numeroCarros**:

```
public static int getNumeroCarros() {
        return numeroCarros;
    }
```

Métodos estáticos ***não podem*** acessar atributos ou métodos de instância (não estáticos) diretamente, eles devem usar uma referência de objeto. Além disso, métodos estáticos não podem usar a palavra-chave **this**, pois não há nenhuma instância para se referir.

4.5 Tipos enumerados (Enums)

Um tipo enumerado (também chamado de enumeração ou **enum**) é um tipo de dados que consiste em um conjunto de constantes. Um exemplo comum de **enum** é: dias da semana. Porque eles são constantes, os nomes dos campos do **enum** devem ser em letras maiúsculas.

Para definir um tipo de **enum** em Java, nós usamos a palavra-chave **enum**. Por exemplo, o seguinte tipo de **enum** define um conjunto de enumerações de cargos:

```
public enum Cargo {
   GERENTE_DE_PROJETOS , LIDER_TECNICO, DIRETOR_GERAL, CEO, CFO;
}
```

Um enum deve ser usado sempre que um conjunto fixo de constantes precisa ser representada.

4.6 Serialização

Serialização é o processo de conversão de um objeto para um formato que pode ser armazenado e depois convertidos de volta mais tarde para um objeto no mesmo ou em outro ambiente.
O Java fornece uma serialização automática que exige que o objeto implemente a interface **java.io.Serializable**. O Java, em seguida, trata a serialização internamente.

O exemplo a seguir é uma classe Java chamada **Funcionario**. É serializável e tem três atributos: um nome, um endereço e o cargo.

```
import java.io.Serializable;
public class Funcionario implements Serializable {

   private String nome;
   private String endereco;
   private Cargo cargo;

   public String getNome() {
      return nome;
   }

   public void setNome(String nome) {
      this.nome = nome;
   }

   public String getEndereco() {
      return endereco;
   }

   public void setEndereco(String endereco) {
      this.endereco = endereco;
   }
```

Agora que temos um objeto serializado, podemos testar o processo de serialização salvando o objeto em um arquivo no disco. O programa a seguir grava um objeto de Funcionario em um arquivo chamado funcionario.ser.

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class TestaSerializacao {
    public static void main (String [] args) {
        Funcionario funcionario = new Funcionario ();
        funcionario.setNome( "Fillipe Cordeiro" );
        funcionario.setEndereco( "São Paulo - SP" );
        funcionario.setCargo(Cargo.CEO);
        try {
            // Para Windows utilize o caminho c:\\funcionario.ser
            FileOutputStream fileOut = new FileOutputStream("/home/desktop/
funcionario.ser");
            ObjectOutputStream out = new ObjectOutputStream (fileOut);
            out.writeObject(out);
            System.out.println("Serializando...");
            out.close();
            fileOut.close();
        } catch (IOException e) {
            e.printStackTrace ();
        }
    }
}
```

Uma vez que você executar o programa, você vai encontrar um arquivo chamado **funcionario.ser** na pasta referenciada.

4.7 Desserialização

Agora nós podemos construir um objeto Funcionario a partir do arquivo que temos no disco, usando um programa totalmente diferente. Tudo o que precisamos é acessar o arquivo salvo funcionario.ser.

O seguinte programa TestaDesserialização, desserializa o objeto Funcionario criado na seção anterior.

```
public class TestaDesserializacao {

   @SuppressWarnings("Unchecked")
   public static void main(String[] args) {
      Funcionario funcionario = new Funcionario();
      try {
         FileInputStream fileIn = new FileInputStream("/home/funcionario.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         funcionario = (Funcionario) in.readObject();
         in.close();
         fileIn.close();
      } catch (IOException i) {
         i.printStackTrace();
         return;
      } catch (ClassNotFoundException c) {
         System.out.println("Classe Funcionario não foi encontrada.");
         c.printStackTrace();
         return;
      } finally {
         if (funcionario instanceof Funcionario) {
            System.out.println("----------------------");
            System.out.println("Desserializado objeto funcionario...");
            System.out.println("Nome:" + funcionario.getNome());
            System.out.println("Endereço:" + funcionario.getEndereco());
            System.out.println("Cargo:" + funcionario.getCargo());
            System.out.println("----------------------");
         }
      }
   }
```

Uma vez que você executar o programa **TestaDesserializacao**, você obterá o seguinte resultado:

```
----------------------
Desserializando objeto funcionario...
Nome: Fillipe Cordeiro
Endereço: São Paulo - SP
Cargo: CEO
----------------------
```

Exercício 4

Objetivos:
- Utilizar métodos e atributos estáticos
- Criação de tipos de enum

Pré-requisitos:
Crie um novo pacote Java no projeto **JavaEssencialAndroid** chamado **br.com.androidpro.javaandroid.exercicio4** antes de resolver esses exercícios.

Exercício:
Criar uma classe Java que representa um objeto de Futebol. Cada futebol é definido por três membros variáveis: volume, peso e cor (onde a cor é apenas limitado a três valores: preto, branco, e azul). Além disso, adicionar uma variável estática à classe, e um método estático que acessa a estática variável.

Crie um pacote Java
1. Na visualização **Project** à esquerda no Intellij IDEA, clique com o botão direito do mouse sobre a pasta **src**.
2. No menu que aparece, vá em **New > Package**.
3. Na caixa de diálogo **"New Package"**, digite no campo o seguinte: **br.com.androidpro.javaandroid.exercicio4**
4. Clique em **OK**

Criar a classe Jogador para escrever seu código
1. Na visualização **Project** à esquerda no Intellij IDEA, clique com o botão direito do mouse sobre o novo pacote.
2. No menu que aparece, vá em **New > Java Class**, e digite **Jogador** no nome.
3. Clique em **OK**

Criar um enum Posicao

1. Na visualização **Project** à esquerda no Intellij IDEA, clique com o botão direito do mouse sobre o novo pacote.
2. No menu que aparece, vá em **New > Java Class**, digite **Jogador** no nome e troque o **Kind** para **Enum**.
3. Clique em **OK**

Escreva o código Java para resolver o exercício

1. Abra o arquivo **Posicao.java**
2. Digite o seguinte código:

```
public enum Posicao {
   ATACANTE, MEIO_CAMPO, DEFENSOR, GOLEIRO;
}
```

3. Abra o arquivo **Jogador.java**
4. Escreva nele o seguinte código:

```
public class Jogador {

   private float peso;
   private float altura;
   private int idade;
   private Posicao posicao;

   public static int quantidaGols = 0;

   public Jogador(float peso, float altura, int idade, Posicao posicao) {
      this.peso = peso;
      this.altura = altura;
      this.idade = idade;
      this.posicao = posicao;
   }

   public static int getQuantidaGols() {
      return quantidaGols;
   }

   public static void setQuantidaGols(int quantidaGols) {
      Jogador.quantidaGols = quantidaGols;
   }
```

```
    public float getPeso() {
        return peso;
    }

    public void setPeso(float peso) {
        this.peso = peso;
    }

    public float getAltura() {
        return altura;
    }

    public void setAltura(float altura) {
        this.altura = altura;
    }

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }

    public Posicao getPosicao() {
        return posicao;
    }

    public void setPosicao(Posicao posicao) {
        this.posicao = posicao;
    }
}
```