Fundamentos da Programação Orientada a Objetos

UNIDADE 06

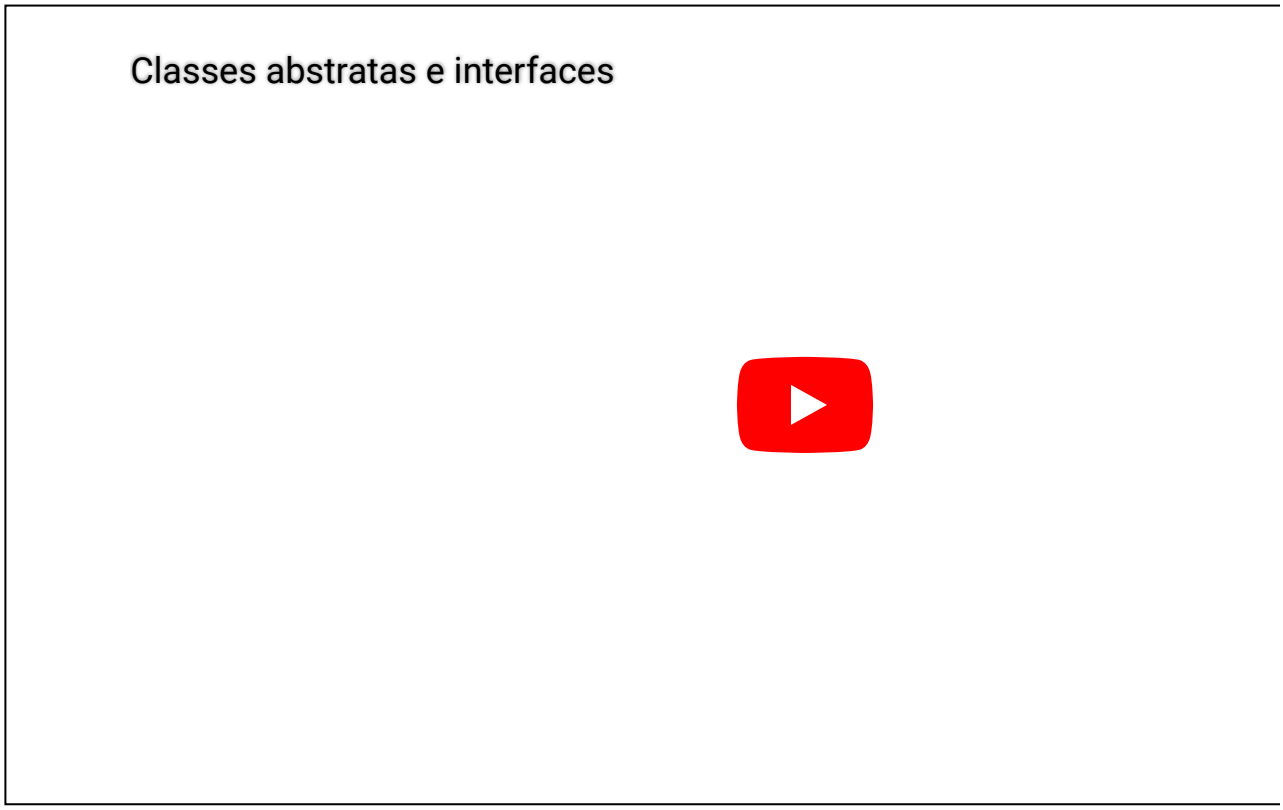
Classes abstratas e interfaces

Até aqui, você já estudou diversos mecanismos que a orientação de objetos fornece para que criemos abstrações (simplificações) do mundo real:

- Com as **classes**, descrevemos em forma de *software* os conceitos do mundo real que estão sendo implementados (ex.: usuários, conta bancária etc.).
- Agrupamos as classes por meio da relação de **associação**, entendendo suas interações ou situações do todo e suas partes.
- Com a **herança**, criamos relações de tipos entre as classes, agrupando-as em hierarquias, de acordo com sua similaridade.
- Com o **encapsulamento**, conseguimos esconder detalhes complexos da implementação, enquanto fornecemos uma **interface pública** de mais simples entendimento.

- Com os **pacotes**, agrupamos classes relacionadas por alguma característica ou função.
- Com o **sobrecarga** e o **polimorfismo** passamos a descrever o conceito de uma operação, mesmo que ela possa ser realizada de formas diferentes.

*Nesta unidade, iremos aprofundar ainda mais o conceito de abstração estudando dois novos temas. O primeiro, são as **classes abstratas** e o segundo o de **interfaces**. Ambos fornecem um mecanismo poderoso para generalizar ainda mais nossos códigos.*



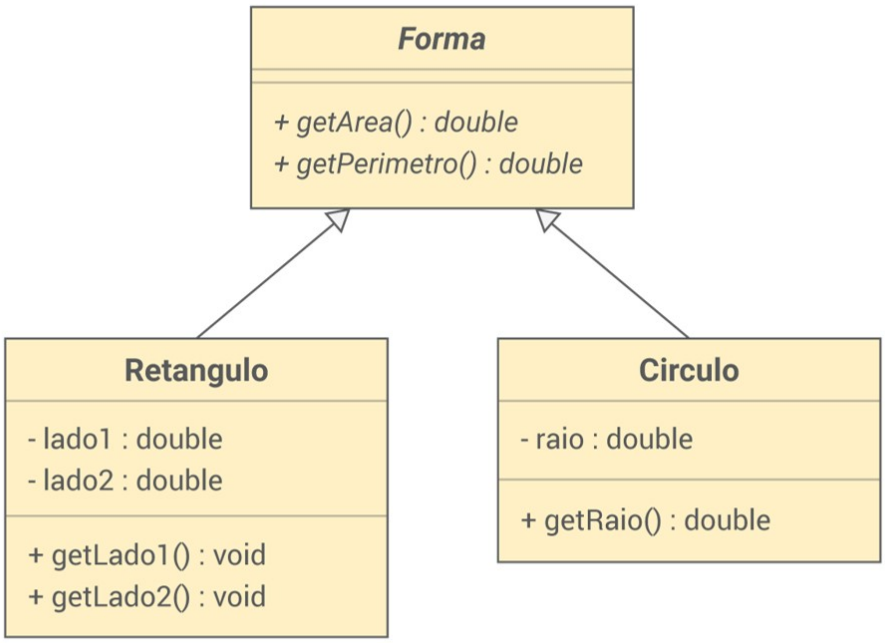
Classes abstratas

EXERCÍCIO

Classes abstratas

1. Implemente a hierarquia de classes **retangulo**, **circulo** e **forma** e execute com o método ***main*** indicado a seguir.

Figura 1 – Exercício: classes abstratas

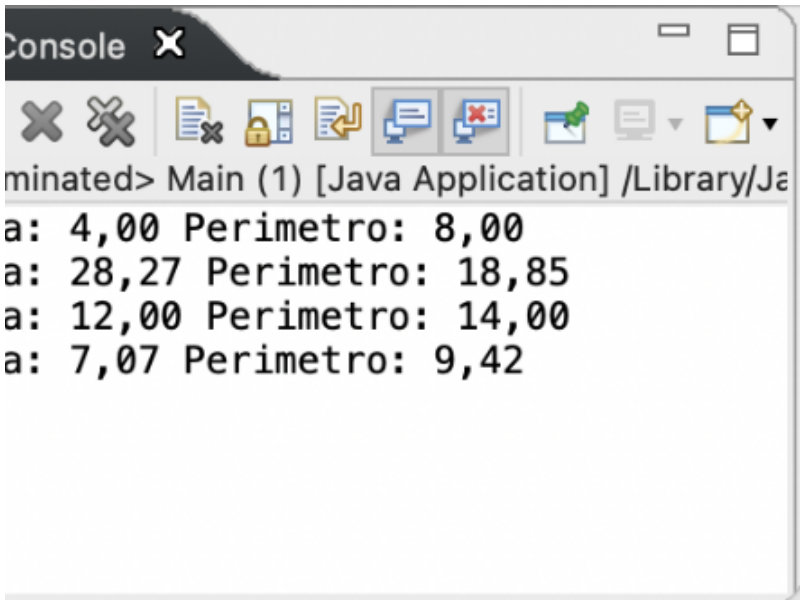


2. Em seguida, responda:

   a. Quais foram as dificuldades na hora de criar os métodos *getArea*() e *getPerimetro*() da classe forma?
   b. Faria sentido instanciar um objeto da classe forma diretamente (ex.: Forma f = new Forma())?

</>

```
import java.util.*;

public class Main {
    public static void
imprimir(List<Forma>
formas) {
        for (Forma
forma : formas) {

System.out.printf("Area:
%.2f Perimetro:
%.2f%n",

forma.getArea(),
forma.getPerimetro());
        }
    }

    public static void
main(String[] args) {
        var formas =
new ArrayList<Forma>();
        formas.add(new
Retangulo(2, 2));
        formas.add(new
Circulo(3));
        formas.add(new
Retangulo(4, 3));
        formas.add(new
Circulo(1.5));

imprimir(formas);
    }
}
```



Fonte: Autores (2021).

Resolução do exercício

Resposta da questão 1 – Classe forma:

```
public class Forma {
    public double getArea() {
        return 0;
    }

    public double getPerimetro() {
        return 0;
    }
}
```

Resposta da questão 1 – Classe circulo:

```

```

```
8
9   public class Circulo extends Forma {
10      private static final double PI =
11  3.1415;
12      private double raio;
13
14      public Circulo(double raio) {
15          this.raio = raio;
16      }
17
18      public double getRaio() {
19          return raio;
20      }
21
22      @Override
23      public double getArea() {
24          return PI * raio * raio;
25      }
26
27      public double getPerimetro() {
28          return 2 * PI * raio;
29      }
30
```

Resposta da questão 1 – Classe Retangulo:

```
</>
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

```
18
19  public class Retangulo extends Forma {
20      private double lado1;
21      private double lado2;
22
23      public Retangulo(double lado1, double
24  lado2) {
25          this.lado1 = lado1;
26          this.lado2 = lado2;
27      }
28
29      public double getLado1() {
30          return lado1;
31      }
32
33      public double getLado2() {
34          return lado2;
35      }
36
37      @Override
38      public double getArea() {
39          return lado1 * lado2;
40      }
41
42      @Override
43      public double getPerimetro() {
44          return (lado1 * 2) + (lado2 * 2);
45      }
46
47  }
48
```

Resposta da questão 2:

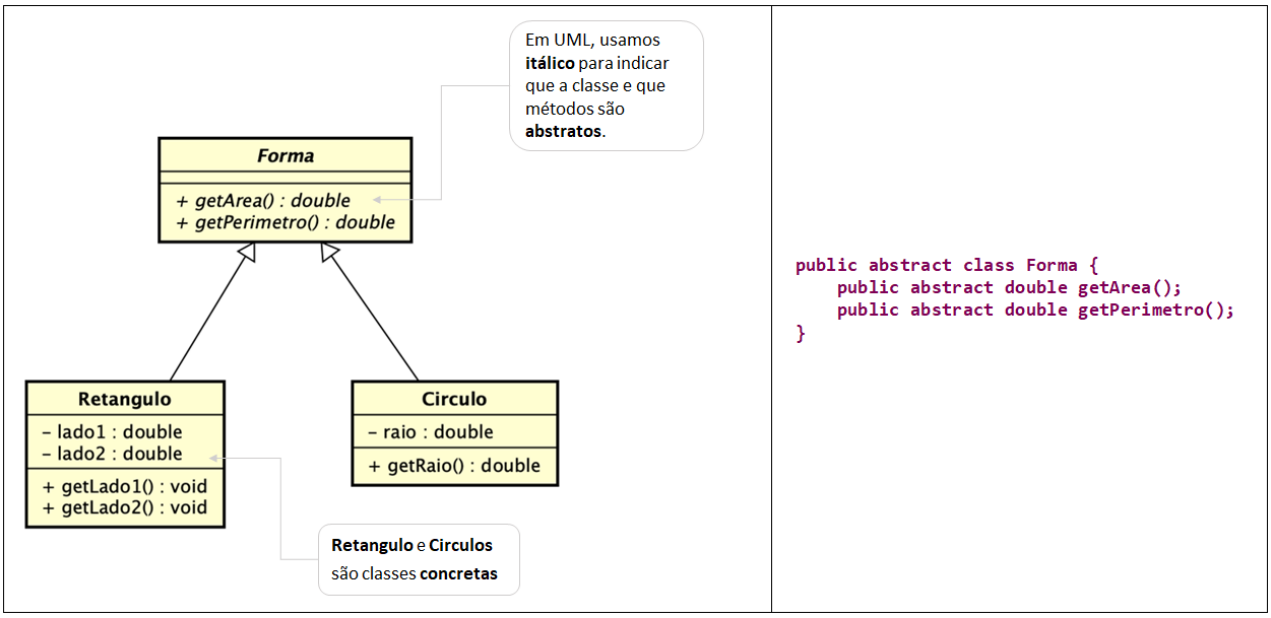
a. Fomos obrigados a retornar 0 nos métodos getArea() e getPerimetro().
b. Não parece fazer sentido.

O exercício traz um exemplo claro da nossa capacidade de **abstração**. Perceba que, enquanto quadrado e circulo são conceitos concretos, e podemos saber exatamente ao que se referem, quando criamos a superclasse os agrupamos em um conceito **abstrato**, o da forma. E o que queremos dizer por abstrato? Nós sabemos que todas as formas têm uma área e um perímetro, mas exatamente como calculá-los só será possível quando estivermos diante de uma forma específica.

Agora, pense um pouco. Se alguém falasse para você: “desenhe uma forma no caderno”, o que você desenharia? Provavelmente, você perguntaria “qual forma?”. Portanto, nós até podemos nos referenciar a círculos ou quadrados como formas, mas somente objetos concretos existem, não uma “forma genérica”.

E como podemos representar essa abstração toda na orientação à objetos? E como implementá-la em Java? Fazemos isso por meio do conceito de **classe abstrata**. Como representa um conceito abstrato, uma classe desse tipo não poderá ser instanciada com a palavra-chave **new**. Além disso, ela pode conter **métodos abstratos**, sem qualquer implementação. Ou seja, indicaremos na classe que sabemos que aquela operação existe para todos os objetos daquele tipo, mas somente as instâncias concretas irão implementá-lo, por meio da sobreposição. Vamos então reescrever a classe forma?

Figura 2 – Classe abstrata forma

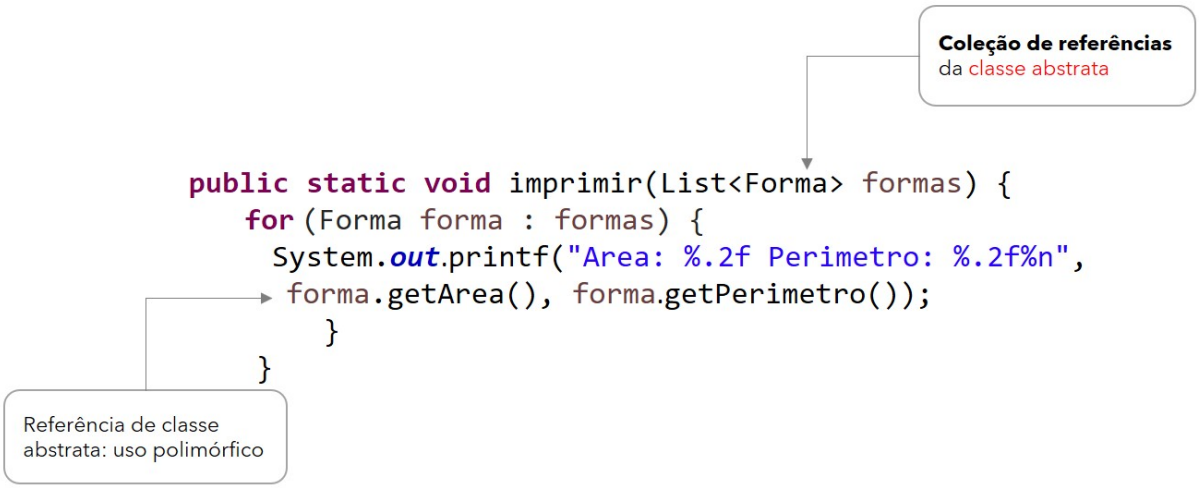


Fonte: Autores (2021).

Qualquer classe concreta que implemente de forma será **obrigada** a sobrescrever **todos** os métodos abstratos, no caso, *getArea()* e *getPerimetro()*. Assim, podemos considerar a classe **forma** como um modelo para a criação de novas classes. Esses métodos poderão

ser usados por meio de polimorfismo, tal como fizemos no método de impressão na classe ***main***:

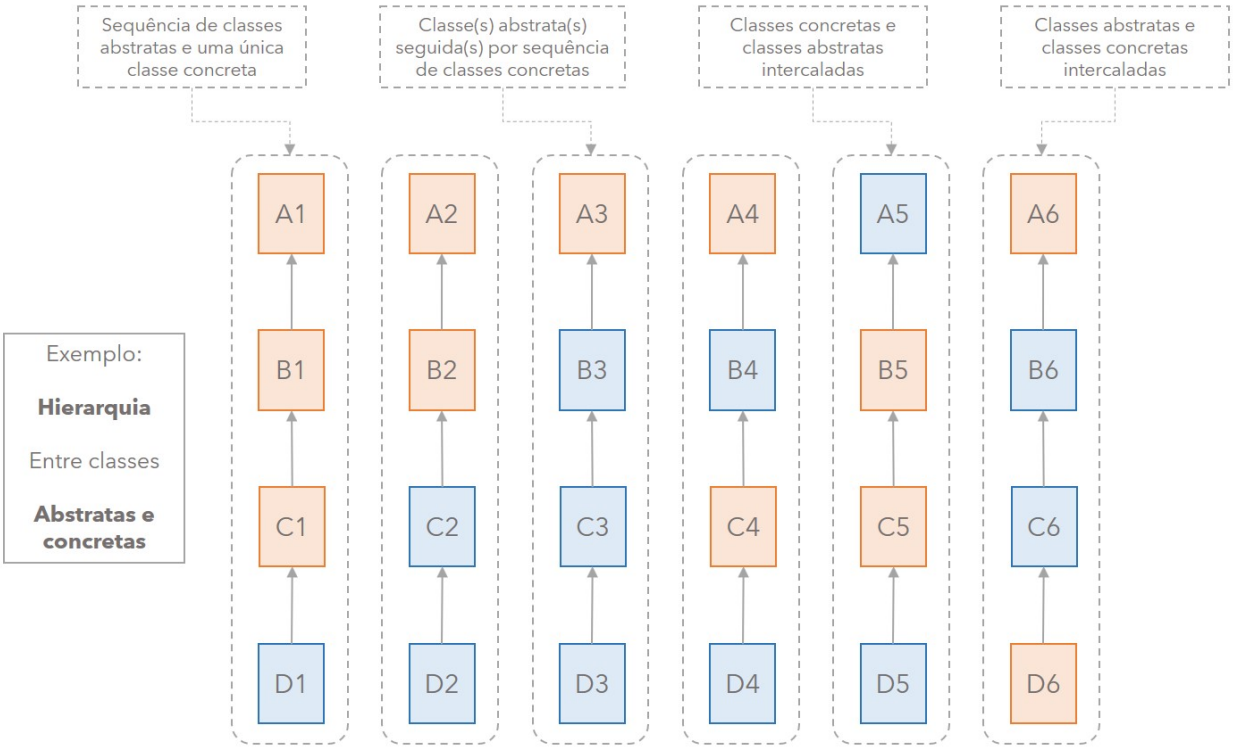
Figura 3 – Hierarquias de classes concretas e abstratas



Fonte: Autores (2021).

Outro detalhe importante é que você pode ter uma classe abstrata que seja filha de outra classe – abstrata ou não. Nesse caso, além de criar outros métodos abstratos, você poderá também deixar alguns métodos abstratos da classe pai sem implementação. O diagrama a seguir resume todas as opções nesse sentido:

Figura 4 – Hierarquias de classes concretas e abstratas



Fonte: Autores (2021).

Por fim, **lembre-se**: **não existe obrigatoriedade** de que uma classe abstrata tenha métodos abstratos.

EXERCÍCIO

Fixação de conceitos: classes abstratas

Para verificar se você entendeu os conceitos, responda às perguntas a seguir.

1. Reflita e responda: um método abstrato pode ser declarado como *private*?
2. Considere as classes declaradas a seguir:

Figura 5 – Exercício: conceitos

```
</>
1   abstract class Pessoa {...}
2   class Estudante extends Pessoa { ... }
3   class Professor extends Pessoa { ... }
4   class EstudanteEspecial extends Estudante
5   { ... }
6
7
```

Quais dessas instanciações são possíveis? Por quê?

a. Pessoa p1 = *new Professor();*
b. Professor prof = *new Professor();*
c. Professor prof2 = *new Estudante();*
d. Pessoa p2 = new Pessoa();
e. Pessoa p3 = new EstudanteEspecial();
f. Estudante e = *new EstudanteEspecial();*

3. As declarações a seguir estão corretas? Justifique sua resposta.

a.

```
</>
1
2
3
```

```
4
5   abstract class Impressora {
6       protected void print() {
7           System.out.println(this);
8       }
9   }
10
```

b.

```
</>
1   abstract class Vazia {}
2
```

c.

```
</>
1   class JogadaDeDados {
2       private int dados;
3       private int faces;
4
5       public abstract int jogar();
6   }
7
```

d.

```
</>
1   abstract class Animal {
2       protected String nome;
3
4       public abstract void fazerSom();
5       protected abstract void respirar();
6       public void imprimir() {
7           System.out.println(nome);
8       }
9   }
10
```

Resolução do exercício

Resposta da questão 1 – Reflita e responda:
Não. Um método abstrato não pode ser private, pois caso ele não seja visível para as sub-classes, não há como as mesmas o sobrescreverem.

Resposta da questão 2 – Instanciações possíveis:
a. Possível: professor é uma classe concreta e filha de pessoa:
`Professor instanceof Pessoa == true`
b. Possível: instanciação de classe concreta
c. Não é possível: professor e estudante no mesmo nível da hierarquia. Um estudante não é um professor, e nem vice-versa.
d. Não é possível: pessoa é uma classe abstrata, não pode ser instanciada.
e. Possível: EstudanteEspecial é uma classe concreta (e descendente de Pessoa)
f. Possível: EstudanteEspecial é uma classe concreta (e descendente de Estudante)

Resposta da questão 3 – Declarações:

a. Correta: uma classe abstrata não precisa conter métodos abstratos.

b. Correta: qualquer classe pode ou não ter métodos ou atributos. Uma classe vazia é perfeitamente válida, abstrata ou não!

c. Incorreta: a classe JogadaDeDados não é abstrata, portanto, o método jogar() não pode ser abstrato. Ele deveria ser implementado, ou a classe deveria ser declarada também como abstrata.

c. Não é possível: professor e estudante estão no mesmo nível da hierarquia. Um estudante não é um professor, e nem vice-versa.

d. Correta: uma classe abstrata pode ter métodos e atributos não abstratos, com qualquer visibilidade. Também podem ter métodos abstratos, desde que sua visibilidade não seja private.

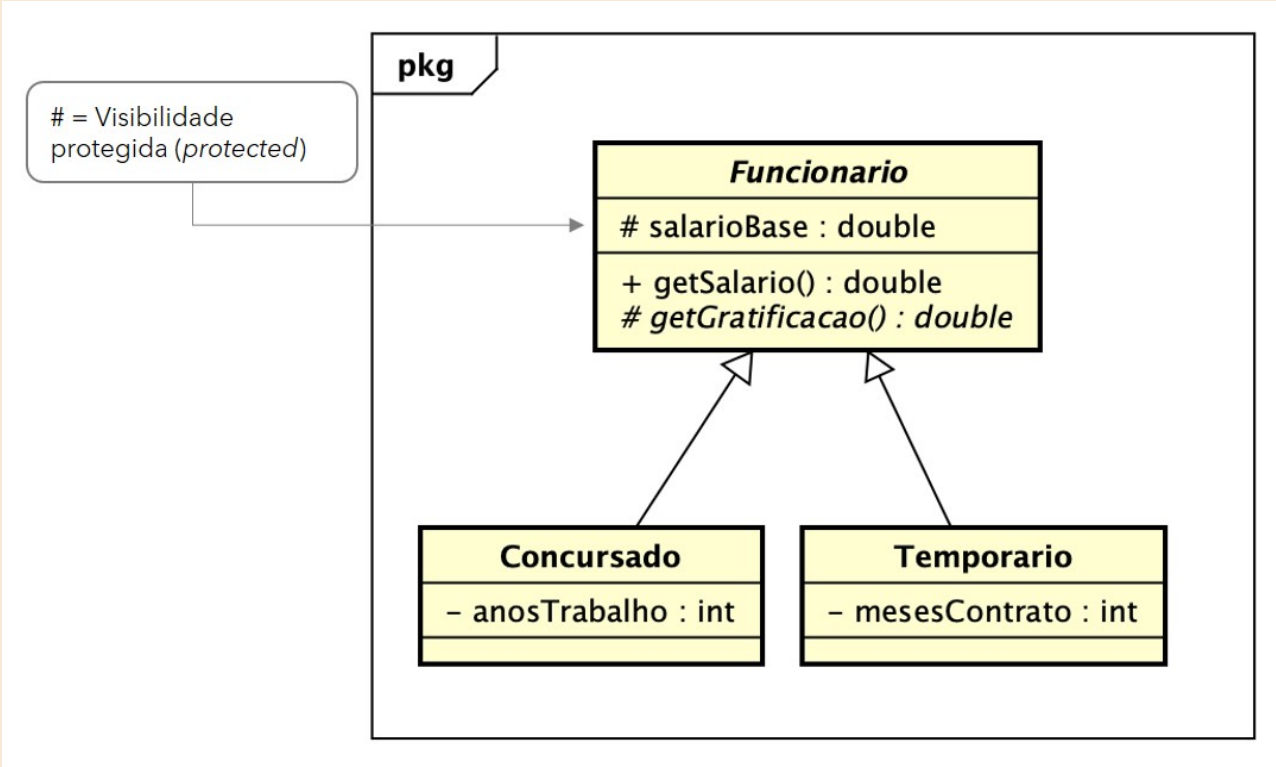
Fonte: Autores (2021).

EXPERIMENTE

Implementação

Fixação de conceitos: classes abstratas

1. Implemente as classes descritas a seguir:



Regras:

a) O cálculo do salário é dado por:

Salário-base + gratificação.

b) A gratificação dos funcionários concursados é:

2% do salário-base a cada cinco anos completos de casa (quinquênio).

c) A gratificação dos funcionários temporários é:

R$ 100,00 para cada mês de contrato.

d) Crie construtores nas classes de modo que seja possível informar o valor de todos os atributos. Ex.: var joao = new Concursado(4000, 5);.

2. Crie uma classe *main* com uma lista de funcionários. Em seguida, crie um método de impressão para listar todos os funcionários:

a. João, concursado, com salário-base de R$ 4000,00 e 5 anos de profissão.
b. José, temporário, com salário de R$ 2000,00 e 12 meses de contrato.
c. Maria, concursada, com salário mensal de 2400,00 e 34 anos de casa.
d. Pedro, temporário, com salário mensal de R$ 1200,00 e 10 meses de contrato.

Quer saber se acertou o exercício? Acesse o protocolo de correção aqui.

Interfaces

Como você estudou, a relação de herança produz uma relação de tipos, entre a classe pai e a filha. Todos os métodos e atributos são herdados. Não é à toa que dizemos que a herança cria uma **relação forte**, pois, como as classes, herdam os atributos e implementações. Isso tem uma implicação importante: uma modificação na classe pai poderá impactar em todas as suas classes filhas – sendo o ideal testar todas as classes para ver se um erro inesperado não foi acidentalmente inserido.

Há, porém, uma alternativa: as interfaces. Elas são similares às classes abstratas, porém com algumas limitações:

1. Não podem ter atributos.
2. Somente podem conter métodos públicos e abstratos.
3. Uma interface pode fazer herança, mas somente com outras interfaces. Porém, a limitação de uma única interface pai não existe.

Criamos as interfaces por meio da palavra-chave **interface**. E declaramos interfaces uma classe que implementa por meio da palavra-chave ***implements***. Uma classe pode implementar qualquer número de interfaces, e isso pode ocorrer em qualquer nível da hierarquia de classes.

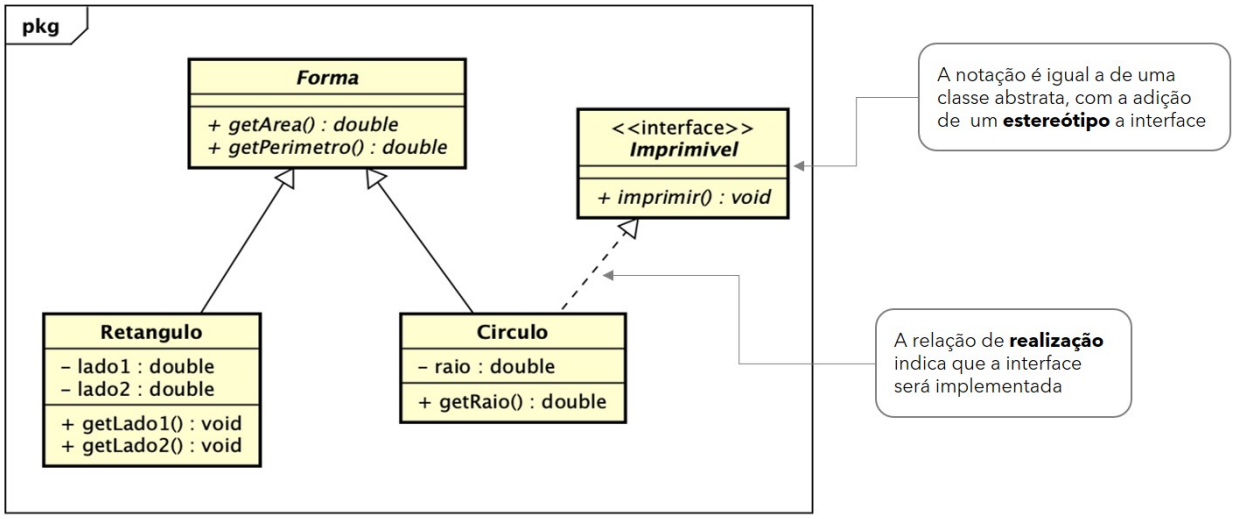
Outras similaridades ainda são observadas:

1. O operador *instanceof* funcionará para interfaces igualmente como funciona para a herança.
2. Todos os métodos abstratos das interfaces também terão de ser sobrepostos nas classes concretas.

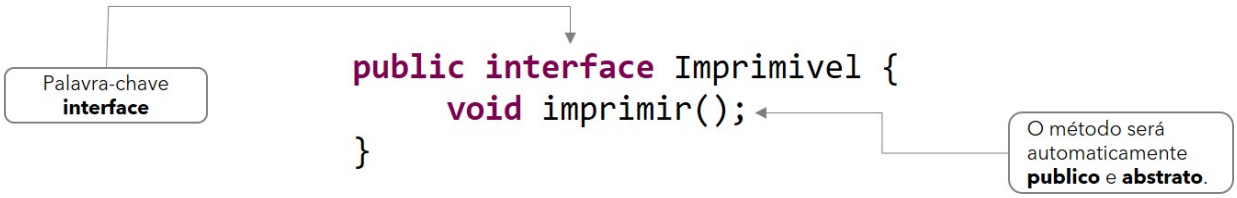
Veja um exemplo completo na Figura 7 a seguir:

Figura 7 – Declaração de uma interface (UML e Java)

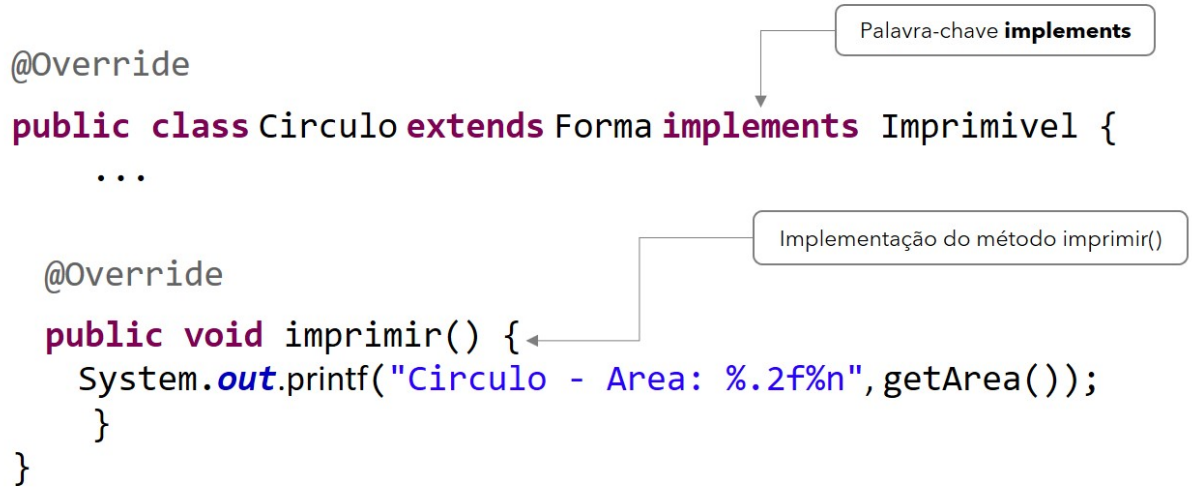
UML



Declaração



Uso



Fonte: Autores (2021).

Conceitualmente, podemos imaginar uma interface como um **contrato** entre a classe e o mundo exterior. A interface define os termos desse contrato, descrevendo qual é o comportamento esperado por aquela classe, enquanto a classe que o implementa compromete-se a cumprir esse contrato.

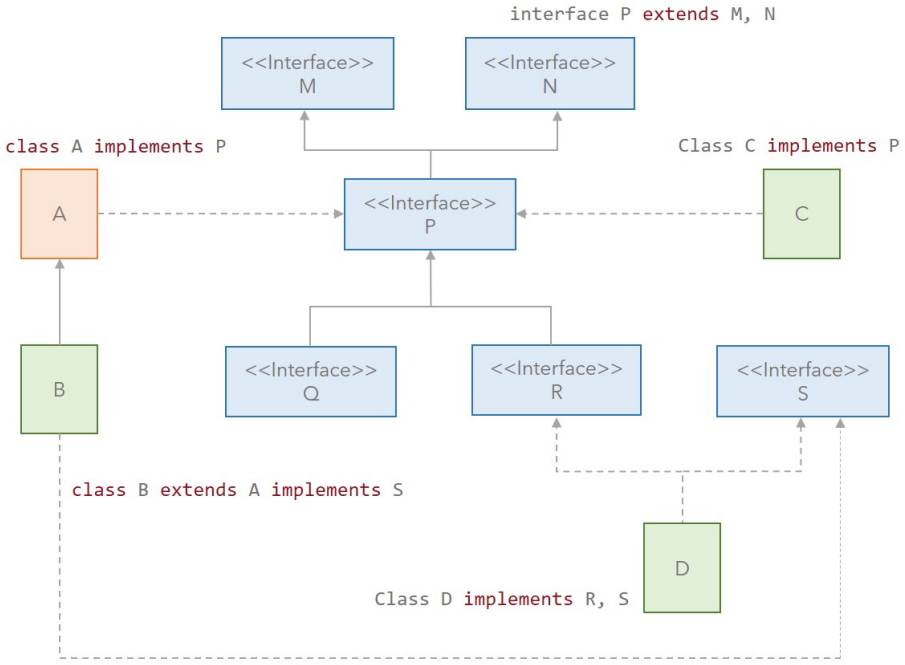
Como não há compromisso com uma implementação, a relação entre a interface e as classes que a implementam é bem mais fraca do que a herança. Sempre que puder, dê preferência ao uso de interfaces no lugar de classes abstratas.

EXERCÍCIO

Interfaces

Considere a hierarquia de classes e interfaces a seguir. Para melhorar a visualização, classes abstratas foram destacadas em rosa, concretas em azul e as interfaces em verde.

Figura 8 – Exercício: interfaces, classes e classes abstratas



Agora, assinale as alternativas corretas, justificando suas respostas:

a. B *instanceof* P retornará verdadeiro.
b. Se C tentasse implementar R, daria erro, uma vez que haveria métodos redundantes entre P e R.
c. A interface P está incorreta, já que você pode dar *implements* em várias interfaces, mas a herança com *extends* só permite um pai.
d. Se um método aceitar um parâmetro do tipo M, poderemos utilizar objetos de qualquer classe descrita no diagrama.

Fonte: Autores (2021).

Resolução do exercício

Resposta da questão 1 – Instanciações possíveis:

a. Correta: como B é filho de A e A implementa a interface P, então B também implementará a interface indiretamente.
b. Incorreta: como os métodos de interfaces não contêm implementação, não há conflitos em casos de redundâncias.
c. Incorreta: uma interface pode ter múltiplas interfaces pai. Apenas classes estão limitadas a estender de um único pai.
d. Correta: todas as classes possuem implementação, direta ou indiretamente, com a interface M.

Controlando a versão do código

Você deve estar notando: os códigos que você escreve estão ficando cada vez mais longos! Antes, eram programas simples, com um único arquivo. Agora, com as classes, o código já se divide em vários arquivos – um por classe – em várias pastas, uma por pacote.

Além disso, à medida que o sistema evolui, criamos funções, movemos trechos de código de lugar, apagamos trechos de código, mudamos regras etc.

Naturalmente, vem o "medo" dessas mudanças. E se uma função que foi apagada voltar a ser necessária? E se um erro for introduzido em uma modificação e precisar voltar atrás? Quem nunca se sentiu tentado a criar um "*zip*" em um determinado momento para "garantir" que aquela versão do código sempre exista? Esse problema se torna ainda pior se você se lembrar que, profissionalmente, você raramente programará sozinho. Como gerenciar o código de uma equipe inteira?

Felizmente, uma solução profissional para esse problema já foi criada. São os **sistemas de controle de versão**. Eles permitem a criação de **repositórios de código**, em que as alterações são controladas.

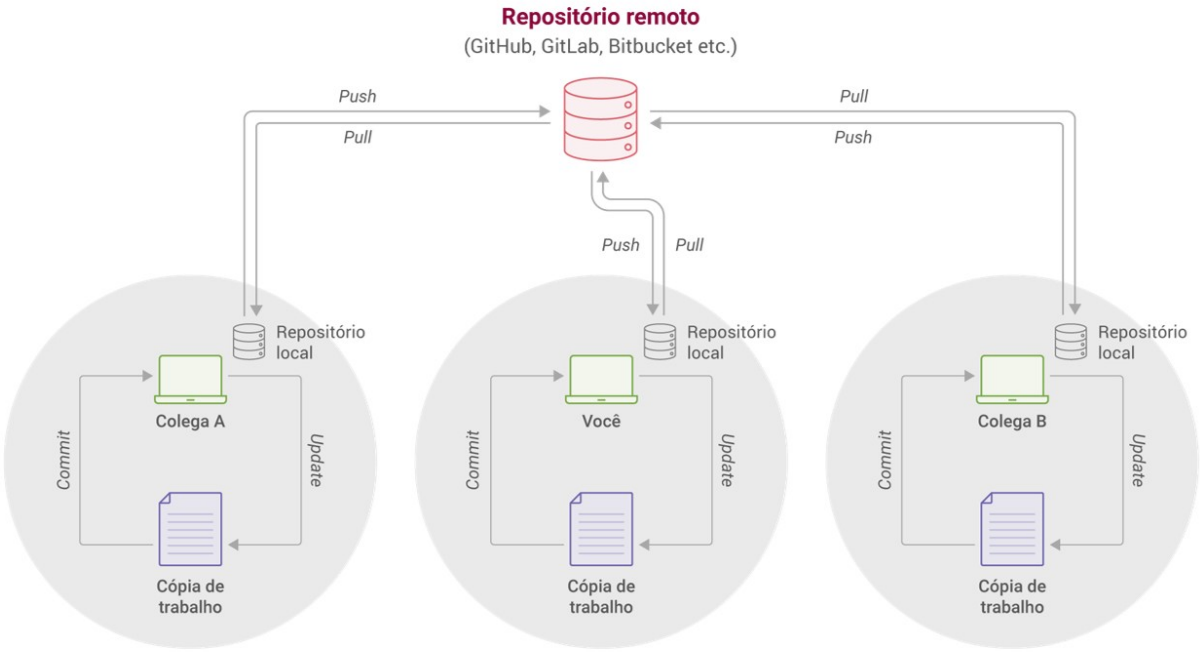
> Os **sistemas de controle de versão** permitem verificar histórico dos arquivos, gerenciar conflitos do código criado em equipe, marcar o código quando uma versão do software for produzida, compartilhar o código na internet, entre outras coisas.

O sistema de controle de versão Git

Um dos sistemas de controle de versão mais populares é chamado de **Git** e desenvolvido criado por Linus Torvalds, criador do sistema operacional Linux. Além de gratuito, o Git possui uma arquitetura distribuída, permitindo que você sincronize um

repositório com outro. Assim, você poderá trabalhar em uma versão do código em sua máquina, enquanto seus colegas fazem o mesmo nas deles, utilizando uma versão local do repositório – com todo conforto que o controle de versão permite. Quando as modificações estão prontas, você poderá enviar todo trabalho para um repositório remoto, normalmente compartilhado entre os membros da equipe. Essa arquitetura é demonstrada na figura a seguir.

Figura 9 – Arquitetura do Git



Fonte: Autores (2021).

Outra vantagem é que o repositório compartilhado pode estar na internet, e há diversos serviços gratuitos que permitem o compartilhamento – público ou não – de seu código (Github, Gitlab, Bitbucket etc.). Isso garantirá não só que você possa trabalhar em equipe, como também já fará o backup do seu programa de forma segura.

Trabalhando com o Git

Em primeiro lugar, o Git é um *software*, feito para operar em linha de comando. É necessário instalá-lo e configurá-lo. Confira os vídeos que acompanham a unidade para isso. Neste tópico, descrevemos os principais conceitos e comandos e porque usá-los, sem entrar em detalhes específicos de instalação ou IDE. Nosso objetivo aqui é que você entenda quais comandos existem, criando um vocabulário básico – que é comum na maior parte dos softwares de controle de versão. Por isso, omitiremos detalhes de alguns comandos mais complexos – que podem ser dados facilmente pela IDE.

Criando repositórios

Todo controle de versão ocorre em repositórios de código. Para criá-los, utilizamos o comando ***git init***. Entretanto, esse comando raramente será usado por você, pois ele criará um repositório somente na máquina local.

A forma mais tradicional de trabalhar é com um repositório remoto, digamos, criado no Github. Nesse caso, ao invés de criar um repositório do zero, você gerar um clone e sincronizá-lo de tempos em tempos.

Para fazer isso, usamos o comando ***git clone***, seguido do nome do repositório. Isso criará na sua máquina local uma pasta, com a cópia do repositório remoto. Por exemplo, para clonar o repositório do código da JVM do OpenJDK, você poderia usar o comando:

```
git clone https://github.com/openjdk/jdk.git openjdk
```

Isso criaria em sua máquina uma pasta chamada *openjdk*, com todo o código do repositório remoto dentro. A pasta criada em sua máquina será o **repositório local**. O repositório do Git de onde você clonou é o **repositório remoto**. O nome do repositório (dado no último parâmetro) é opcional e, se omitido, criará a pasta com o mesmo nome do repositório remoto (jdk, nesse caso).

Todo código modificado, salvo no disco, compõe ainda um terceiro conceito, chamado **cópia de trabalho.** Isso porque você poderá restaurar um arquivo modificado na sua cópia de trabalho para a versão que está em seu repositório local a qualquer momento.

Modificando arquivos no repositório

Todos os arquivos em um repositório podem ser controlados ou não controlados. Os arquivos controlados são aqueles que o Git está monitorando e saberá se foram ou não modificados, ou estão prestes a serem adicionados ao repositório (*staged*).

Obviamente, você fez o clone de um repositório, todos os arquivos clonados em seu interior serão controlados e não modificados. Quando você modifica um arquivo, ele passará para o estado de arquivo modificado. E, quando você criar um arquivo novo, ele não será controlado. Sua IDE mostrará esses status de maneira visual.

Para que um arquivo seja controlado, é necessário utilizar o comando ***git add***. Isso trocará a arquivo criado para um estado chamado de ***staged***. Isto é, ele será adicionado ao repositório local assim que as modificações forem confirmadas. Você pode utilizar o comando de duas formas:

- ***git add*** **<nomeArquivo>**: para adicionar um único arquivo.
- ***git add .***: para adicionar todos os arquivos não controlados.

Para excluir um arquivo, é possível utilizar o comando ***git rm***. Ele excluirá o arquivo do disco, mas essa exclusão também será considerada ***staged***. Isto é, ela também precisará ser confirmada.

E como confirmamos essas modificações? Por meio do comando ***git commit***. Ele não só enviará as informações ao repositório local, como também permitirá que você associe a elas uma mensagem de *commit*. Trata-se de um texto, no qual você descreve de maneira sucinta as modificações que fez. É esse texto que será exibido caso você queira ver o histórico das suas modificações.

Mantendo arquivos não controlados

Algumas vezes, você poderá ter arquivos não controlados que deseja manter apenas em seu disco local, mas não os enviar para o repositório. Alguns exemplos de arquivos desse tipo seriam:

- Um arquivo desse tipo poderia ser o xml, em que estão as configurações do banco de dados que a aplicação usará, e que será diferente para cada desenvolvedor da equipe.
- Outra possibilidade seriam os arquivos específicos da sua IDE favorita, que não se referem ao código da aplicação em si.
- Os arquivos binários gerados pela JVM (.class, .java) e que podem ser recompilados por qualquer desenvolvedor.

Para evitar que um arquivo seja controlado, criamos um arquivo chamado *.gitignore* e colocamos lá dentro o nome do arquivo ou pastas que queremos ignorar. Isso evitará que ele seja adicionado caso o *git add .* seja utilizado.

Há, inclusive, uma variação do comando git rm, útil para quando queremos remover um arquivo do repositório, mas não do disco – por exemplo, um arquivo que você havia esquecido de adicionar ao *.gitignore*. É por meio da opção *--cached*, por exemplo:

```
git rm --cached README
```

Revertendo alterações

Às vezes, você começou a editar um arquivo e se arrependeu, e gostaria de retorná-lo a versão que está atualmente em seu repositório. Para isso, deve utilizar o comando de ***revert***. Ele exigirá o código do último *commit*, o que pode ser trabalhoso. Felizmente, sua IDE conseguirá obter essa informação automaticamente.

Enviando arquivos ao repositório remoto

Por fim, você pode usar os comandos ***git push*** e ***git pull*** para enviar ou receber arquivos do repositório remoto.

Note que você pode trabalhar no repositório local por bastante tempo, antes de dar um *push*. Isso fará com que o *git* agrupe todas as modificações necessárias e as envie de maneira muitíssimo eficiente. Na verdade, sua eficiência foi um dos atributos que o tornou tão popular.

Para onde ir a partir daqui

Obviamente, há muito mais o que estudar sobre o Git. Confira o material em vídeo desta unidade, no qual abordamos o Git de maneira prática, utilizando o Netbeans.

Caso você queira se aprofundar, há também ótimos materiais disponíveis na internet:

- Livro Pro-git, disponível gratuitamente online em: https://git-scm.com/.
- Guia prático e simplificado: https://rogerdudler.github.io/git-guide/index.pt_BR.html.

Por fim, saiba que o esforço de aprender o Git **vale a pena**! Não só você terá muito mais tranquilidade ao codificar, como essa é uma ferramenta padrão na indústria. Você não entrará em uma empresa hoje em dia que não utilize um controle de versão e, muito provavelmente, será o Git.

EXPERIMENTE

Instalando o Git

Conhecendo o GitHub

Trabalhando com o Git no Eclipse

Trabalhando com o Git no IntelliJ

Referências

GODOY, V. **Programação orientada a objetos I**. Curitiba: IESDE, 2019.

HORSTMANN, C. S.; CORNELL, G. **Core Java – volume I**. 8. ed. São Paulo: Pearson, 2010.

SCHILDT, H. **Java para iniciantes**. Porto Alegre: Bookman, 2015.