easy java Magazine
Edição 57

DESAFIO:
Introdução ao TomEE na prática
Desenvolva aplicações Java EE com o servidor de aplicações da Apache

Orientação a Objetos: uma visão inicial
Conheça a história e os primeiros passos para desenvolver com esse paradigma

DEVMEDIA

THREADS:

UMA NOVA PERSPECTIVA

Dos primeiros recursos às novidades do Java 8

ISSN 2179625-4

Edição 57 • 2016 • ISSN 2173625-4

EXPEDIENTE

Editor
Eduardo Spínola (*eduspinola@gmail.com*)

Consultor Técnico Diogo Souza (diogosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse **www.devmedia.com.br/central**, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site Easy Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:

EDUARDO OLIVEIRA SPÍNOLA
eduspinola.wordpress.com
@eduspinola / @Java_Magazine

Sumário

Orientação a Objetos: Uma visão inicial

Conheça neste artigo um pouco da história sobre a orientação a objetos e os primeiros passos para desenvolver sistemas com esse paradigma

No desenvolvimento de sistemas tratamos a orientação a objetos como um paradigma de programação, ou seja, como uma forma de se implementar um código. No entanto, este é um tema com aplicação em diversas áreas e por este motivo as literaturas sobre o assunto podem apresentar definições às vezes divergentes. Devido ao nosso escopo, obviamente, neste artigo vamos tratar da orientação a objetos direcionada ao desenvolvimento de software e, portanto, a estudaremos como: um conceito da engenharia de software no qual os elementos de uma solução são representados como objetos.

Ampliando nossos horizontes, podemos dizer que a orientação a objetos vem antes mesmo da engenharia, ou até da tecnologia. Os mecanismos que o cérebro humano utiliza para pensar e ver o mundo são totalmente orientados a objetos. Percebemos isso ao notar que categorizamos e agrupamos os elementos em nossa percepção para poder compreendê-los, exatamente como fazemos em uma análise orientada a objetos.

Nesse contexto, para representar os elementos identificados durante a análise de um sistema que segue esse paradigma, é necessário conhecermos um pouco sobre modelagem, de modo que saibamos como construir, de forma visual, os componentes da solução através de notações e diagramas padronizados.

Antes disso, no entanto, com o intuito de ter um melhor entendimento da evolução do desenvolvimento de software que nos trouxe até aqui, vamos contar um pouco da história da programação e as tecnologias que surgiram para dar suporte ao paradigma orientado a objetos. Posteriormente, abordaremos uma notação comum para representar os componentes de uma solução OO, a UML, e logo após, analisando os principais conceitos referentes a objetos, vamos defini-los, apresentando sua notação em UML e sua respectiva codificação em Java.

Fique por dentro

A orientação a objetos é um assunto fundamental para todos que irão trabalhar com desenvolvimento, principalmente para quem vai atuar com Java ou qualquer linguagem que adote esse paradigma. Portanto, se você está começando a se aventurar no mundo da programação ou dando os primeiros passos com o Java, este artigo será bastante útil por apresentar os conceitos necessários para desenvolver orientado a objetos. Como importante complemento, conheceremos também alguns recursos básicos da linguagem de modelagem UML, opção mais utilizada para criar diagramas que representam um sistema orientado a objetos.

Por fim, faremos uma reflexão sobre a função da orientação a objetos na TI hoje em dia e os caminhos que ela tende a seguir no futuro com o advento de novos paradigmas para o desenvolvimento e modelagem de soluções, ponderando se esta irá acompanhar as novas tendências ou se será substituída por elas.

Notação UML

Agora que entendemos como surgiu a orientação a objetos, vamos começar a falar da modelagem dos sistemas que são desenvolvidos com esta tecnologia. Para representar um software devemos utilizar uma notação que seja capaz de dar o entendimento ao programador do que deve ser implementado com a linguagem de programação. E para ser eficaz, essa notação também precisa ser compreendida pelos analistas, responsáveis por identificar e modelar os requisitos do que será entregue como solução. No âmbito dos sistemas orientados a objetos a opção mais utilizada é a UML.

A criação da notação UML teve início em outubro de 1994, quando Rumbaugh se juntou a Booch na Rational Software Corporation. Nesta época o constante crescimento da Rational fez com que surgisse a necessidade de organizar e definir as etapas de desenvolvimento adotadas em seus projetos. Foi concebido,

assim, o processo de desenvolvimento unificado, denominado RUP (*Rational Unified Process*). Contudo, como os sistemas implementados por outras empresas careciam de um processo mais formal e definido, logo o mercado também abraçou o RUP.

A partir disso, começou a surgir a necessidade de uma notação igualmente formal para representar a modelagem das soluções. No começo dos anos 1990, as notações existentes eram o Booch, o OMT e o OOSE. Como cada solução apresentava vantagens e desvantagens, não havia um consenso de mercado sobre qual usar, até que, aproveitando o melhor de cada notação, em outubro de 1995, a Rational lançou a versão 0.8 do Unified Method.

Em 1995, fizeram o escopo do projeto sobre modelagem da Rational ser expandido para incorporar mais alguns diagramas, já presentes na notação OOSE. Nasceu, então, em junho de 1996, a versão 0.9 da UML, e, em 1997, finalmente a Linguagem de Modelagem Unificada foi aprovada como padrão pelo OMG (*Object Management Group*).

No entanto, a aceitação e difusão do processo unificado da Rational, bem como o envolvimento de seus profissionais na criação da notação UML, fez com que esta fosse constantemente confundida com um processo de desenvolvimento. UML, entretanto, não é uma metodologia de desenvolvimento de software, mas sim um modo de visualizar a interação dos componentes de software que farão parte do projeto. Em outras palavras, Unified Modeling Language é uma linguagem de modelagem de diagramas que permite especificar, visualizar e documentar modelos de software, incluindo detalhes de sua estrutura, design e da comunicação entre os objetos que compõem o sistema.

Atualmente na versão 2.5, a UML define 23 diagramas para representar não só a modelagem das classes de uma solução, mas também diversos outros pontos importantes para o processo de desenvolvimento e entrega de um software, como casos de uso, implantação, comportamento, colaboração, entre outros.

Dentre tantos diagramas o mais utilizado é o diagrama de classes. Nele representamos não apenas classes, como indicado em seu nome, mas também objetos e seus relacionamentos, e por isso é tão importante que o desenvolvedor Java tenha um bom domínio sobre ele. Pensando nisso, nos próximos tópicos nos tornaremos mais familiarizados com o diagrama de classes a partir da conceituação dos principais elementos da orientação a objetos e da sua representação em UML e em código Java.

Orientação a Objetos

A orientação a objetos é um paradigma de programação que define um sistema através da interação e composição de diversas unidades chamadas objetos. Este conceito é utilizado naturalmente pela mente humana para categorizar tudo no mundo real, o que aproxima a programação orientada a objetos da forma como pensamos. É comum, por exemplo, ao visualizar e compreender os elementos do nosso cotidiano, os identificarmos através de suas características, interações e composição dos mesmos. Assim, quando entramos em uma biblioteca repleta de livros, iremos identificar o grupo de livros que tratam do assunto que queremos para chegar ao conteúdo desejado. Ou ainda, quando nos socializamos com um grupo de pessoas, vamos identificar quais delas podem se tornar nossas amigas ou não, quais são solteiras ou casadas, entre outras possibilidades.

Nota

Apesar de inúmeras vantagens, os principais diagramas UML são repletos de notações e símbolos técnicos que nem sempre são compreendidos pelos perfis negociais (principalmente pelos clientes). Assim, muitas vezes o cliente não consegue entender completamente a solução através destes diagramas. Devido a isso, notações como BPM e BPEL têm surgido para preencher essa lacuna, oferecendo opções de modelagem mais próximas dos perfis negociais e aproximando mais os clientes, analistas e desenvolvedores.

Observamos, com isso, que a mente humana funciona de forma orientada a objetos. Para exemplificar mais uma vez esse conceito, note que quando entramos em uma sala de aula, automaticamente procuramos detectar quantas cadeiras, janelas, computadores ela tem. Ao agrupar objetos com características em comum (cadeiras, por exemplo) para quantificá-los e compreendê-los, estamos fazendo uma análise orientada a objetos.

Perceber isso evidencia um dos grandes diferenciais da orientação a objetos: a redução do tempo necessário para se compreender e modelar um problema. Afinal, programar mais próximo da forma que pensamos naturalmente diminui o tempo de entendimento de uma solução.

A partir disso, para iniciar nossos estudos sobre a orientação a objetos, vamos nos aprofundar na definição de cada elemento desse paradigma. E para cada elemento, iremos formalizar a notação UML utilizada para modelá-lo e a sintaxe Java básica para expor com código o conceito aprendido.

O nosso objetivo com isso é que consigamos pensar o nosso sistema com o raciocínio orientado a objetos, modelar e formalizar seus requisitos através da notação UML e, enfim, possamos implementar em Java o que foi definido através de diagramas.

Afinal, o que é um objeto?

Para entender como funciona a interação entre objetos, devemos primeiro conhecer o que é um objeto. Na literatura, podemos encontrar várias definições a respeito. Aqui, no entanto, vamos nos ater à área de programação. De um modo geral, um objeto é uma unidade de software que é utilizada para representar alguma coisa existente no mundo real. Exemplos de objetos no mundo real são uma árvore, um carro, um cachorro, etc. Em um software, objeto é o componente do sistema que desempenha a função do objeto no mundo real.

Quando olhamos à nossa volta, podemos encontrar muitos objetos dos mais diversos tipos e para diferenciar e classificar estes objetos, levamos em consideração duas características: seu estado e seus comportamentos. O estado é representado pelas características do objeto, como: para um objeto do tipo **pessoa**, podemos citar como características o nome, a idade, a altura, entre outros. Os comportamentos, por sua vez, são representados pelas coisas que o objeto pode fazer, como estudar, correr, trabalhar, no caso do objeto **pessoa**. Dito isso, para entender como o objeto armazena e adquire esses comportamentos, vamos nos aprofundar um pouco mais nos próximos tópicos.

O que é uma classe?

Ao observar vários objetos é possível verificar que mesmo sendo diferentes eles possuem muitas semelhanças, isto é, várias características que utilizamos para quantificar, identificar e qualificar são as mesmas. Tomemos como exemplo, mais uma vez, os objetos do tipo **Pessoa**. Certamente existem várias pessoas das mais diferentes raças e credos, mas algumas características todas elas devem ter, como nome, idade e capacidade de raciocínio mais elevado do que outros seres vivos. Sendo assim, podemos dizer que todo ser que tem as características descritas são um tipo de pessoa, ou ainda, um objeto do tipo **Pessoa**.

A classe é o molde a partir do qual objetos são criados. Para compreender melhor essa afirmação, nos analisemos como objetos representados em uma solução de software: você é um objeto, ou ainda, você é uma instância (ou uma materialização) de uma classe; neste caso, a classe **Pessoa**. No linguajar da orientação a objetos, dizemos que você é uma instância da classe **Pessoa**.

À vista disso, uma classe pode ser composta por dois tipos elementos: atributos, ou seja, características que podem ser modificadas, aumentadas ou diminuídas, mas que estarão presentes em todo objeto dessa classe; e métodos, isto é, comportamentos, que podem ser implementados das mais diversas formas, mas que todo objeto da classe **Pessoa** também terá.

A partir disso, podemos definir classe como um conceito que representa objetos com características semelhantes e que especifica os estados desses objetos através de atributos e os comportamentos através de métodos.

Na **Figura 1** é apresentada a classe **Pessoa** em sua representação em UML e na **Listagem 1** temos sua implementação em Java. Neste momento é importante salientar que para atuar como um bom desenvolvedor é fundamental possuir a habilidade de "ler" um diagrama UML e, a partir deste, implementar o código Java.

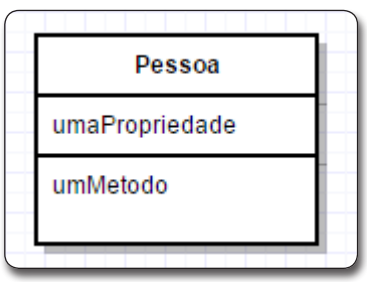

Figura 1. Classe Pessoa em UML

Listagem 1. Código da classe Pessoa.

```
//Declara uma classe pública chamada Pessoa
public class Pessoa {

   //Atributos e métodos da classe
   public String umaPropriedade;
   public void umMetodo(){}

}
```

Com essa implementação pronta, podemos criar (instanciar) objetos da classe **Pessoa**. Para isso, observe o código descrito na **Listagem 2**. Este se trata de uma classe Java simples com o método **main()**, responsável por executar o programa, instanciando um objeto **Pessoa**.

Listagem 2. Código da classe de teste.

```
public class Teste {

   //Método utilizado para iniciar o programa
   public static void main(String... args){
      //Instanciando um objeto da classe pessoa
      Pessoa obj = new Pessoa();
      //Invocando métodos do objeto instanciado (obj)
      obj.umMetodo();
   }

}
```

O que é um atributo?

Os atributos são as características da classe, os locais onde os estados dos objetos são armazenados, sendo também chamados de campos ou propriedades. Eles são responsáveis por guardar o estado do objeto, isto é, as características que podem ser mensuradas através de valores.

Tomando como exemplo os objetos da classe **Pessoa**, características como nome, idade e endereço serão guardadas em atributos. O conjunto dos valores desses atributos compõe o estado do objeto. Na **Figura 2** é apresentada a modelagem da classe **Pessoa** com os atributos **nome**, **idade** e **pai** e o comportamento **umMetodo** (sobre o qual falaremos no próximo tópico).

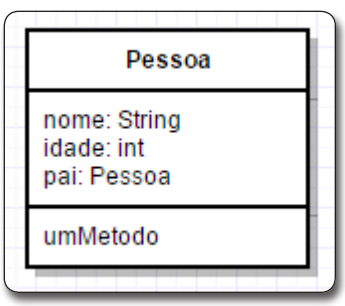

Figura 2. Classe pessoa com atributos e métodos representada em UML

Agora vamos desenvolver um pouco mais o código da classe **Pessoa** para demonstrar como codificar os atributos do diagrama. O resultado é apresentado na **Listagem 3**.

Na **Listagem 4** temos um programa teste que demonstra como criar um objeto e manipular seus atributos.

Ao implementar os atributos é possível defini-los como campos que permitem ou não a alteração de seus valores. Os atributos que têm a capacidade de ter seus valores alterados são chamados de variáveis, e os atributos que mantêm seus valores inalterados são chamados de constantes ou atributos finais.

Repare, em nosso código, que o acesso aos atributos é feito através de métodos.

Essa abordagem é comum na orientação a objetos e se chama encapsulamento (para mais informações a respeito, leia o artigo "Encapsulamento em Java: Primeiros passos", publicado na easy Java Magazine 43). A partir dessa prática podemos centralizar o acesso a um atributo e isso poderá facilitar, dentre outras coisas, o desenvolvimento de validações ou formatações de dados.

Listagem 3. Código da classe Pessoa com atributos.

```
public class Pessoa {

   //Atributos da classe pessoa
   private String nome;
   private int idade;
   private Pessoa pai;

   //Métodos de acesso, também chamados de getters e setters
   public String getNome() {
      return nome;
   }
   public void setNome(String nome) {
      this.nome = nome;
   }
   public int getIdade() {
      return idade;
   }
   public void setIdade(int idade) {
      this.idade = idade;
   }
   public Pessoa getPai() {
      return pai;
   }
   public void setPai(Pessoa pai) {
      this.pai = pai;
   }

}
```

Listagem 4. Código da classe de teste utilizando atributos.

```
public class Teste {

   public static void main(String... args){

      //Criando uma instância de objeto da classe pessoa
      Pessoa obj = new Pessoa();

      //Acessando e modificando os atributos da pessoa indiretamente
      obj.setIdade(30);
      obj.setNome("João Felipe");

      //Imprimindo o nome e a idade da pessoa
      System.out.print(obj.getNome() + " tem " + obj.getIdade() + " anos.");
   }

}
```

Objetos com essa estrutura – atributos privados cujo acesso é encapsulado através de métodos de acesso – são chamados de *JavaBeans*, um padrão bastante utilizado por diversos frameworks Java. Por isso, apesar de no nosso dia a dia não implementarmos cada método *get* e *set* de nossas classes, visto que a maioria das IDEs é capaz de gerar esse código, é importante compreender e saber codificar esse padrão, pois em alguns cenários será preciso modificar o código de ao menos um desses métodos.

É comum, por exemplo, no método *get*, termos um código de inicialização caso o atributo esteja nulo ou não preenchido.

Voltando à declaração dos atributos da classe **Pessoa**, observe que a instrução começa com um modificador de acesso chamado **private**. Através desse recurso podemos definir no Java em quais momentos um atributo pode ou não pode ser visualizado, isto é, podemos especificar se o atributo será visível somente na própria classe (caso do modificador **private**), dentro do mesmo pacote ou de qualquer lugar.

Ao declarar os atributos de sua classe, tenha em mente que o Java possui quatro modificadores de acesso: **public**, **protected**, **private** e **default**. Para seguir as melhores práticas de encapsulamento, relacionamento entre classes, abstração e outras é importante conhecer as regras de cada um deles.

O que é um método?

Uma classe, além de estados, pode possuir comportamentos, isto é, métodos. Os métodos representam as ações ou procedimentos que os objetos podem ter. No caso da classe **Pessoa**, os métodos vão definir as ações que uma pessoa pode fazer, por exemplo: pensar, caminhar, ler, entre outras. Assim, se ler é uma característica de **Pessoa**, o método **ler()** deve ser declarado nesta classe; e o modo de execução da leitura, descrito (implementado) neste método.

A partir dessa definição, vale ressaltar que métodos e atributos não estão isolados dentro de uma classe, e muitas vezes podem ter uma forte relação entre si. Por exemplo, a classe **Pessoa** pode ter um atributo do tipo inteiro (**raciocinio**) e esse atributo influenciar diretamente no comportamento do método **pensar()**.

Na **Figura 3** é possível observar a modelagem da classe **Pessoa** e na **Listagem 5** implementamos um comportamento que simula a execução de um pensamento (código do método **pensar()**) utilizando a saída de texto da linha de comando. Note que o atributo **raciocinio** define quantos pensamentos a pessoa irá produzir.

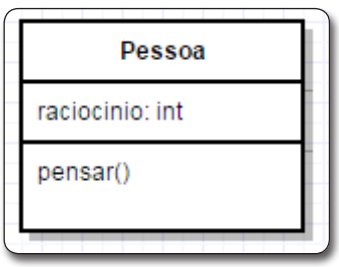

Figura 3. Classe Pessoa com o método pensar() em UML

Para entender melhor a forma como um método é declarado (codificado) e executado, vamos observar alguns pontos dessa listagem. Primeiro, deve-se observar que o código dentro do método **pensar()** não é orientado a objetos, mas sim estruturado, ou seja, as instruções são executadas uma após a outra obedecendo as estruturas de controle de fluxo. Como citamos no começo do artigo, a linguagem Java possui trechos de código estruturado, apesar de ser orientada a objetos.

Diante do conteúdo exposto, podemos definir método como o comportamento que um objeto possui.

No exemplo apresentado na **Listagem 6**, a classe de testes irá criar objetos do tipo **pessoa** e chamar (invocar) seus métodos **setRaciocinio()** e **pensar()** para executar algumas ações.

Listagem 5. Código da classe Pessoa com o método pensar().

```
public class Pessoa {
  //Característica (atributo) de uma pessoa
  private int raciocinio;

  //Comportamento de uma pessoa
  public void pensar(){
    for (int i = 0; i < raciocinio; i++) {
      System.out.println("pensamento ");
    }
  }

  public int getRaciocinio() {
    return raciocinio;
  }

  public void setRaciocinio(int raciocinio) {
    this.raciocinio = raciocinio;
  }
}
```

Listagem 6. Código da classe de testes demonstrando o uso de métodos.

```
public class Teste {
  public static void main(String... args){
    //pessoa que terá 1 pensamento
    Pessoa pessoa1 = new Pessoa();
    pessoa1.setRaciocinio(1);
    pessoa1.pensar();

    //pessoa que terá 10 pensamentos
    Pessoa pessoa2 = new Pessoa();
    pessoa2.setRaciocinio(10);
    pessoa2.pensar();
  }
}
```

Na linguagem Java, além de implementar os comportamentos do objeto, os métodos podem ou não retornar algum valor ao término de sua execução. No código da **Listagem 5**, a palavra reservada **void** especifica que o método não irá retornar nenhum valor.

Instruções como essa, que modificam o comportamento de um método, são comuns em diversas linguagens. Na **Listagem 7**, por exemplo, apresentamos como esses modificadores são usados em Java.

A seguir, descrevemos, de forma breve, alguns dos recursos dessa linguagem que permitem enriquecer a implementação dos métodos:

- **Modificadores de acesso:** Semelhantes aos modificadores de acesso dos atributos, definem quando e onde os métodos podem ser vistos e chamados;
- **Especificação de Retorno:** Permite a um método especificar o tipo de objeto que será retornado por ele, ou que nada será retornado;
- **Generics:** Inserido no Java 1.5, permite não declarar diretamente o tipo de retorno ou dos parâmetros, especificando para isso tipos genéricos, o que possibilita que o mesmo método seja executado com tipos diferentes de atributos ou retorne também objetos de diversos tipos;
- **Anotações:** Inserido no Java 1.5, representa metadados dos métodos e pode adicionar diversos comportamentos personalizados. É um recurso presente não só em frameworks comuns do cenário corporativo (como Hibernate, Spring, entre outros), como também em especificações da própria API (como JPA e EJB); e
- **Modificadores de comportamento:** Permitem alterar as configurações de compilação, implementação e execução do método possibilitando, por exemplo, desenvolver código nativo da plataforma de execução, definir o método como transacional ou sincronizado, especificá-lo como um método que não pode ser alterado ou ainda um método sem implementação (método abstrato).

Listagem 7. Classe com exemplos de métodos.

```
public abstract class ExemploMetodos {
  //Método que não poderá ser sobrescrito pelas subclasses
  public final void metodoFinal() {}
  //Método que deve ter sua implementação definida pelas subclasses
  public abstract void metodoAbstrato();
  //Método que só atenderá a uma execução por vez
  public synchronized void metodoSincronizado() {}
}
```

O que é Herança?

Nos tópicos anteriores exploramos os conceitos relacionados a classes, atributos e métodos. Contudo, a orientado a objetos não se resume a isso. Para a construção de um sistema OO, dentre outras coisas, é comum a definição de várias classes que precisam estar conectadas para viabilizar as funcionalidades do sistema. Uma das formas de estabelecer essas conexões é através da herança.

De forma objetiva, podemos dizer que herança é a solução que temos para especificar, ou especializar, classes através de subclasses que podem adicionar e/ou alterar comportamentos de sua superclasse. Tendo isso em mente, note que existem diversos cenários nos quais precisamos de uma classe um pouco mais específica, como acontece quando lidamos com um controle de estoque e temos produtos específicos que compartilham características de um produto mais genérico, como preço e quantidade.

Para fixar esta ideia, analisemos o exemplo a seguir: ao aprimorar mais uma vez o código da classe **Pessoa**, suponha que vamos tornar nosso código mais específico para poder classificar as pessoas em alguns tipos (desenvolvedor, atleta e jornalista) e assim poder tratá-las de forma diferenciada. Para simplificar, limitamos nosso escopo a três tipos.

Como todos eles são especializações de pessoa, todos respiram, pensam e andam, correto? Porém, cada um apresenta também características próprias. Por exemplo: todos podem falar, mas certamente um jornalista tem uma maior facilidade em expressar suas ideias do que um atleta, que por sua vez deve correr mais rápido do que um desenvolvedor, que por fim deve ter um raciocínio lógico mais bem elaborado do que os outros dois. Mesmo assim, note que todos continuam sendo pessoas. Para desenvolver um código que represente esse cenário criamos as classes **Desenvolvedor**, **Atleta** e **Jornalista**; classes estas que, utilizando a palavra-chave **extends**, irão herdar os comportamentos de **Pessoa** e, assim, ter a possibilidade de definir

comportamentos particulares, como analisar um sistema ou redigir um texto, ou ainda modificar comportamentos herdados, provendo implementações diferentes de métodos já existentes, como o **Atleta**, que poderá correr mais que uma **Pessoa** convencional.

Na **Figura 4** temos a representação em UML do cenário descrito e na **Listagem 8**, o código que o implementa.

Como podemos verificar, a herança é uma ótima forma de reutilizar código. Para validar essa afirmação, observe que todas as especializações de **Pessoa** possuem três comportamentos, mas cada uma implementou apenas um. Sem herança seria necessário implementar nove métodos, três em cada classe, fora a dificuldade de manutenção e compreensão devido à repetição.

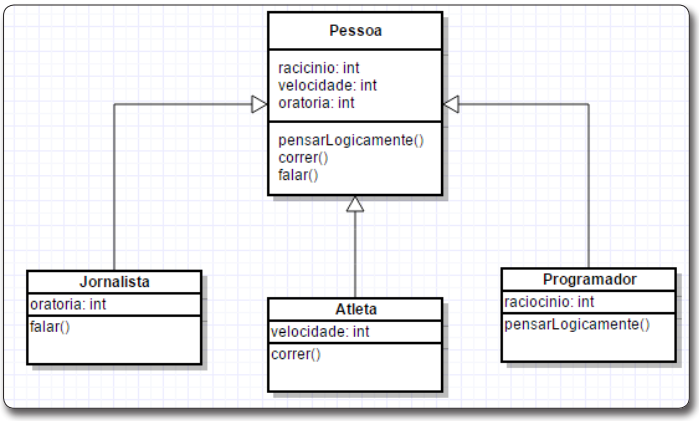

Figura 4. Diagrama de classes com as classes Programador, Atleta e Jornalista, subclasses de Pessoa

O que é uma associação?

Outra forma bastante comum das classes se relacionarem é associando umas às outras através de atributos. Também conhecida como agregação, agregação compartilhada e composição, a associação basicamente define uma das formas pelas quais uma classe pode se relacionar com outra.

Para demonstrar esse conceito na prática, suponha que o objeto do tipo **Aluno** possui relacionamentos com **Curso** e **Pessoa**. A associação entre **Aluno** e **Curso** é estabelecida através do aluno que participa de um curso e o curso que possui um ou muitos alunos. Já a associação entre **Aluno** e **Pessoa** retrata um relacionamento de herança, visto que **Aluno** é um tipo **Pessoa**. Note que herança também é classificada como um tipo de associação.

A partir disso, na **Figura 5** temos uma representação em UML dessa proposta e na **Listagem 9** implementamos o código desse diagrama. Nesse código, como esperado, o relacionamento entre **Aluno** e **Pessoa** é uma herança, definida através da declaração de **extends**. Por sua vez, a classe **Curso** tem uma associação com **Aluno**, representada através de uma lista de objetos do tipo **Aluno**.

Tecnologias e referências

O que é uma tecnologia sem uma boa ferramenta para utilizá-la? Para potencializar o uso da orientação a objetos, existem boas ferramentas tanto para modelagem quanto para desenvolvimento.

Hoje, para a modelagem orientada a objetos, o padrão UML é o mais consolidado e difundido, sendo aceito e inclusive solicitado como requisito básico para trabalhar em grande parte dos projetos que adotam tal paradigma.

Como diferencial, o UML é mantido por uma organização filiada à OMG, o que dá certa garantia de que a notação continuará acompanhando as novas tendências, bem como terá uma padronização no uso e interpretação pelas ferramentas de mercado. Para consultar o material de referência sobre UML, veja o endereço indicado na seção **Links**.

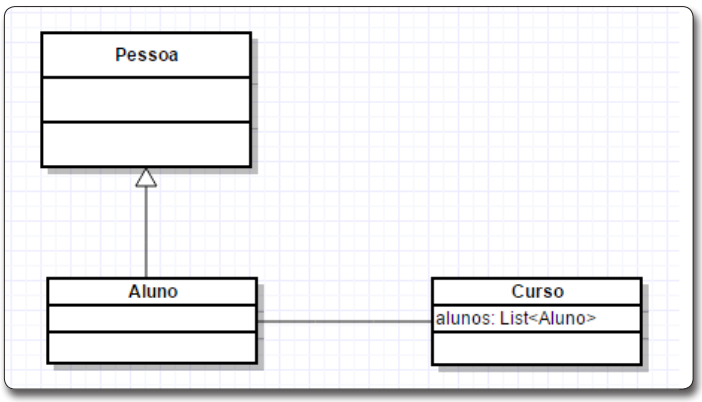

Figura 5. Representação em UML da associação de Aluno e Curso e da herança de Aluno e Pessoa.

Para construir diagramas UML existem diversas ferramentas. Dentre elas, algumas pagas fornecem como diferencial a geração de código a partir dos diagramas e trabalho colaborativo. Ademais, devido à sua alta adoção, é possível, até mesmo, encontrar sites que permitem construir diagramas on-line. Este é o caso do Gliffy e do Creately, utilizado para criar os diagramas UML deste artigo e que ainda viabilizam a modelagem em diversas outras notações mais atuais, como BPM.

Como ferramenta de referência para modelagem UML, a mais utilizada ainda é o ASTAH (vide seção **Links**), que veio para substituir o JUDE, solução gratuita (que posteriormente passou a contar também com uma versão paga (*Professional*)) usada por vários anos em grandes projetos que descartaram softwares caros – que tinham o apoio de grandes corporações – para optar pelo simples, objetivo e sem exagerados custos de licença.

Com relação aos primeiros passos no mundo do desenvolvimento, é fundamental falar também das IDEs (*Integrated Development Environment*). No caso do Java, as mais utilizadas são o Eclipse e o NetBeans, soluções gratuitas e que fornecem um grande leque de recursos que possibilitam programar com qualidade, segurança e respeitando algumas boas práticas, principalmente relacionadas à organização do código. O endereço para download destas ferramentas pode ser encontrado na seção **Links**.

A orientação a objetos facilita bastante a compreensão e representação das soluções tecnológicas em código, por mais complexas que estas possam ser. Desde longa data (talvez desde a criação do COBOL) as linguagens de programação se aproximam da forma como pensamos. Neste sentido, a orientação a objeto representa, hoje, o auge deste objetivo.

Utilizando-a, podemos representar exatamente, ou com bastante semelhança, a forma como a mente humana vê o mundo: separando e categorizando para depois agrupar.

Outro aspecto importante a se observar é que os sistemas têm evoluído rapidamente não apenas em complexidade negocial, atendendo requisitos cada vez mais avançados, mas também em complexidade tecnológica, o que pode ser verificado no desenvolvimento para web, para dispositivos móveis, no armazenamento em nuvem e agora também com a IoT. Assim, com uma frequência cada vez mais alta surgem novas formas de desenvolver e também de implantar ou integrar sistemas, além de novas metodologias de interação entre usuários, analistas e desenvolvedores.

Dentro deste cenário, existem os que digam que a orientação a objetos não acompanhou a evolução tecnológica e será substituída por novas arquiteturas e metodologias, como SOA ou BPM. Porém, se observarmos as definições destas tecnologias de forma mais aprofundada, poderemos notar que as mesmas não excluem a orientação a objetos, mas sim trazem uma evolução e novas formas de utilizá-la. Para confirmar isso, basta uma lida mais aprofundada nos conceitos e paradigmas mais recentes para identificarmos que ideias como herança, encapsulamento e abstração continuam presentes.

Enfim, a orientação a objetos é um conceito fundamental para poder atuar com as principais tecnologias de mercado, de tal forma que entendê-la à fundo é um grande diferencial entre o programador comum e o desenvolvedor diferenciado, capaz de implementar e manter códigos com qualidade.

Listagem 8. Código das classes que representam as heranças da classe Pessoa.

```
public class Pessoa {
   //Características (atributos) de uma pessoa
   public int raciocinio = 10;
   public int velocidade = 10;
   public int oratoria = 10;

   //Comportamentos de uma pessoa
   public void pensarLogicamente(){}
   public void correr(){}
   public void falar(){}

   //Métodos get e set dos atributos
}

public class Atleta extends Pessoa {
   //Sobrescrevendo o valor do atributo velocidade
   //O atleta possui velocidade maior do que uma pessoa comum
   public int velocidade = 100;

   //Sobrescrita do comportamento correr() usando o atributo velocidade
   public void correr(){
      System.out.println("Metros percorridos: " + velocidade);
   }
}

public class Jornalista extends Pessoa {
   //Sobrescrevendo o valor do atributo oratoria
   //O jornalista possui oratória melhor do que uma pessoa comum
   public int oratoria = 100;

   //Sobrescrita do comportamento discursar() usando o atributo oratoria
   public void correr(){
      System.out.println("Capacidade de oratória: " + oratoria);
   }
}

public class Programador extends Pessoa {
   //Sobrescrevendo o valor do atributo raciocinio
   //O programador possui raciocinio melhor do que uma pessoa comum
   public int raciocinio = 100;

   //Sobrescrita do comportamento pensar logicamente usando o atributo
   //raciocinio
   public void pensarLogicamente (){
      System.out.println("Raciocínios lógicos feitos: " + raciocinio);
   }
}
```

Listagem 9. Implementação dos relacionamentos entre Aluno, Pessoa e Curso.

```
//Um aluno é uma pessoa, ou seja, a classe Aluno estende a classe Pessoa
public class Aluno extends Pessoa {}


public class Curso {

   //O curso possui alunos
   public List<Aluno> alunos;

}
```

Autor

João Felipe D'Assenção Faria
jfelipe.web@gmail.com
Consultor Java, SOA e BPM. Com 15 anos de carreira, já participou de grandes projetos em diversas empresas e estados, tendo vasta experiência em todas as áreas de projetos empresariais. Certificado Oracle como programador Java, desenvolvedor web e de negócios e especialista SOA e BPM. Apaixonado por filosofia, música e computação.

Links:

Página da OMG sobre UML 2.5.
http://www.omg.org/spec/UML/2.5/

Ferramenta online de modelagem utilizada para desenhar os diagramas deste artigo.
http://www.gliffy.com

Creately - Outra opção para modelagem online.
http://www.creately.com

Página da ASTAH, com diversas ferramentas para modelagem.
http://astah.net

Página da Oracle para download da Java Virtual Machine.
http://www.oracle.com/technetwork/java/javase/downloads/index.html

Eclipse – Ambiente Integrado para Desenvolvimento.
http://www.eclipse.org/

NetBeans – Outra opção de IDE.
https://netbeans.org/

Threads: paralelizando tarefas com os diferentes recursos do Java

Saiba como criar threads utilizando desde os recursos da primeira versão até as opções fornecidas pelo Java 8 e garanta mais desempenho ao seu software

A plataforma Java disponibiliza diversas APIs para implementar o paralelismo desde as suas primeiras versões, e estas veem evoluindo até hoje, trazendo novos recursos e frameworks de alto nível que auxiliam na programação. No entanto, deve-se lembrar que a tecnologia não é tudo. É importante, também, conhecer os conceitos desse tipo de programação e boas práticas no desenvolvimento voltado para esse cenário.

O processamento paralelo, ou concorrente, tem como base um hardware multicore, onde dispõe-se de vários núcleos de processamento. Estas arquiteturas, no início do Java, não eram tão comuns. No entanto, atualmente já se encontram amplamente difundidas, tanto no contexto comercial como doméstico. Diante disso, para que não haja desperdício desses recursos de hardware e possamos extrair mais desempenho do software desenvolvido, é recomendado que alguma técnica de paralelismo seja utilizada.

Como sabemos, existem diversas formas de criar uma aplicação que implemente paralelismo, formas estas que se diferem tanto em técnicas como em tecnologias empregadas. Em vista disso, no decorrer deste artigo serão contextualizadas as principais APIs Java, desde as threads "clássicas" a modernos frameworks de alto nível, visando otimizar a construção, a qualidade e o desempenho do software.

Fique por dentro

Este artigo aborda o paralelismo de tarefas em softwares Java, prática de desenvolvimento muito utilizada e necessária nos dias de hoje, onde temos arquiteturas computacionais essencialmente paralelas. Os desenvolvedores que desejam otimizar o desempenho de um software encontrarão neste artigo explicações sobre o funcionamento das principais APIs multithreaded da plataforma Java, bem como os problemas mais triviais e formas de como evitá-los.

Arquitetura Multicore

Uma arquitetura multicore consiste em uma CPU que possui mais de um núcleo de processamento. Este tipo de hardware permite a execução de mais de uma tarefa simultaneamente, ao contrário das CPUs singlecore, que eram constituídas por apenas um núcleo, o que significa, na prática, que nada era executado efetivamente em paralelo.

A partir do momento em que se tornou inviável desenvolver CPUs com frequências (GHz) mais altas, devido ao superaquecimento, partiu-se para outra abordagem: criar CPUs multicore, isto é, inserir vários núcleos no mesmo chip, com a premissa base de dividir para conquistar.

Ao contrário do que muitos pensam, no entanto, os processadores multicore não somam a capacidade de processamento, e sim possibilitam a divisão das tarefas entre si. Deste modo, um processador de dois núcleos com clock de 2.0 GHz não equivale

a um processador com um núcleo de 4.0 GHz. A tecnologia multicore simplesmente permite a divisão de tarefas entre os núcleos de tal forma que efetivamente se tenha um processamento paralelo e, com isso, seja alcançado o tão almejado ganho de performance.

Contudo, este ganho é possível apenas se o software implementar paralelismo. Neste contexto, os Sistemas Operacionais, há anos, já possuem suporte a multicore, mas isso somente otimiza o desempenho do próprio SO, o que não é suficiente. O ideal é cada software desenvolvido esteja apto a usufruir de todos os recursos de hardware disponíveis para ele.

Ademais, considerando o fato de que hoje já nos deparamos com celulares com processadores de quatro ou oito núcleos, os softwares a eles disponibilizados devem estar preparados para lidar com esta arquitetura. Desde um simples projeto de robótica a um software massivamente paralelo para um supercomputador de milhões de núcleos, a opção por paralelizar ou não, pode significar a diferença entre passar dias processando uma determinada tarefa ou apenas alguns minutos.

Multitasking

O multitasking, ou multitarefa, é a capacidade que sistemas possuem de executar várias tarefas ou processos ao mesmo tempo, compartilhando recursos de processamento como a CPU. Esta habilidade permite ao sistema operacional intercalar rapidamente os processos ativos para ocuparem a CPU, dando a impressão de que estão sendo executados simultaneamente, conforme a **Figura 1**.

No caso de uma arquitetura singlecore, é possível executar apenas uma tarefa por vez. Mas com o multitasking esse problema é contornado gerenciando as tarefas a serem executadas através de uma fila, onde cada uma executa por um determinado tempo na CPU. Nos sistemas operacionais isto se chama escalonamento de processos.

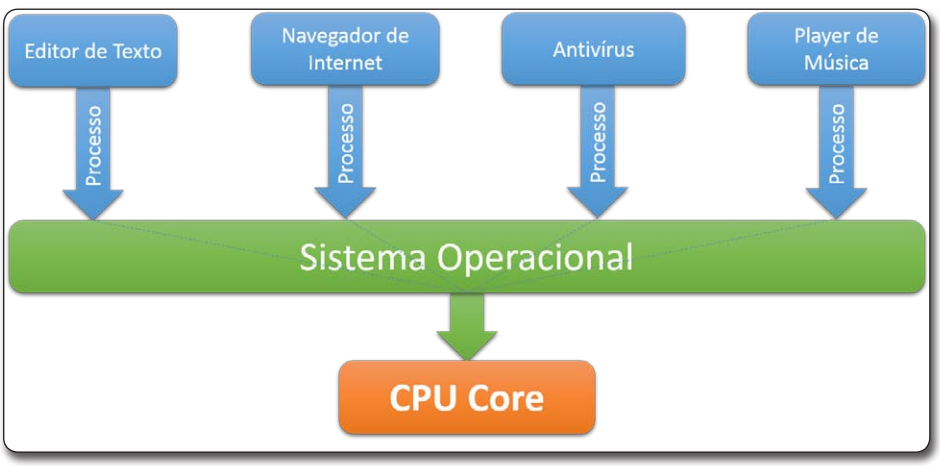

Figura 1. Processos executando em um núcleo

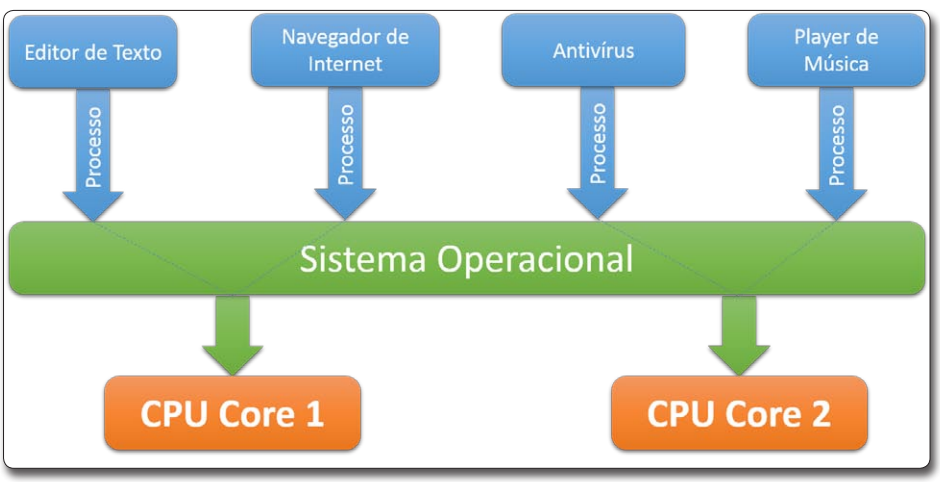

Figura 2. Arquitetura multicore executando processos

Em arquiteturas multicore, efetivamente os processos podem ser executados simultaneamente, conforme a **Figura 2**, mas ainda depende do escalonamento no sistema operacional, pois geralmente temos mais processos ativos do que núcleos disponíveis para processar.

Desta forma, mais núcleos de processamento significam que mais tarefas simultâneas podem ser desempenhadas. Contudo, vale ressaltar que isto só é possível se o software que está sendo executado sobre tal arquitetura implementa o processamento concorrente. De nada adianta um processador de oito núcleos se o software utiliza apenas um.

Multithreading

De certo modo, podemos compreender multithreading como uma evolução do multitasking, mas em nível de processo. Ele, basicamente, permite ao software subdividir suas tarefas em trechos de código independentes e capazes de executar em paralelo, chamados de threads. Com isto, cada uma destas tarefas pode ser executada em paralelo caso haja vários núcleos, conforme demonstra a **Figura 3**.

Diversos benefícios são adquiridos com este recurso, mas, sem dúvida, o mais procurado é o ganho de performance. Além deste, no entanto, também é válido destacar o uso mais eficiente da CPU. Sabendo dessa importância, nosso próximo passo é entender o que são as threads e como criá-las para subdividir as tarefas do software.

Threads

Na plataforma Java, as threads são, de fato, o único mecanismo de concorrência suportado. De forma simples, podemos entender esse recurso como trechos de código que operam independentemente da sequência de execução principal. Como diferencial, enquanto os processos de

software não dividem um mesmo espaço de memória, as threads, sim, e isso lhes permite compartilhar dados e informações dentro do contexto do software.

Cada objeto de thread possui um identificador único e inalterável, um nome, uma prioridade, um estado, um gerenciador de exceções, um espaço para armazenamento local e uma série de estruturas utilizadas pela JVM e pelo sistema operacional, salvando seu contexto enquanto ela permanece pausada pelo escalonador.

Na JVM, as threads são escalonadas de forma preemptiva seguindo a metodologia "*round-robin*". Isso quer dizer que o escalonador pode pausá-las e dar espaço e tempo para outra thread ser executada, conforme a **Figura 4**. O tempo que cada thread recebe para processar se dá conforme a prioridade que ela possui, ou seja, threads com prioridade mais alta ganham mais tempo para processar e são escalonadas com mais frequência do que as outras.

Também é possível observar na **Figura 4** que apenas uma thread é executada por vez. Isto normalmente acontece em casos onde só há um núcleo de processamento, o software implementa um sincronismo de threads que não as permite executar em paralelo ou quando o sistema não faz uso de threads. Na **Figura 5**, por outro lado, temos um cenário bem diferente, com várias threads executando paralelamente e otimizando o uso da CPU.

Desde seu início a plataforma Java foi projetada para suportar programação concorrente. De lá para cá, principalmente a partir da versão 5, foram incluídas APIs de alto nível que nos fornecem cada vez mais recursos para a implementação de tarefas paralelas, como as APIs presentes nos pacotes **java.util.concurrent.***.

Saiba que toda aplicação Java possui, no mínimo, uma thread. Esta é criada e iniciada pela JVM quando iniciamos a aplicação e sua tarefa é executar o método **main()** da classe principal. Ela, portanto, executará sequencialmente os códigos contidos neste método até que termine, quando a thread encerrará seu processamento e a aplicação poderá ser finalizada.

Em Java, existem basicamente duas maneiras de criar threads:

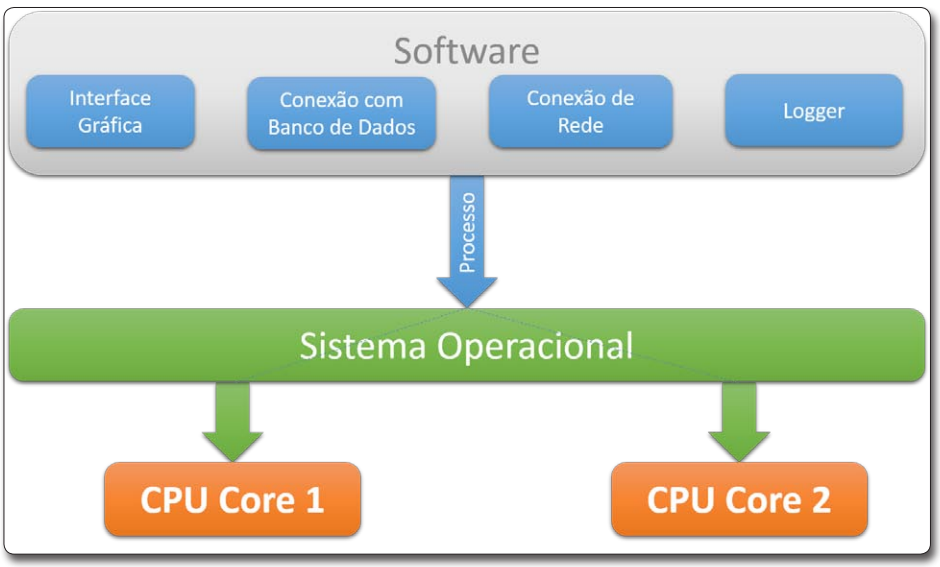

Figura 3. Processo executando várias tarefas

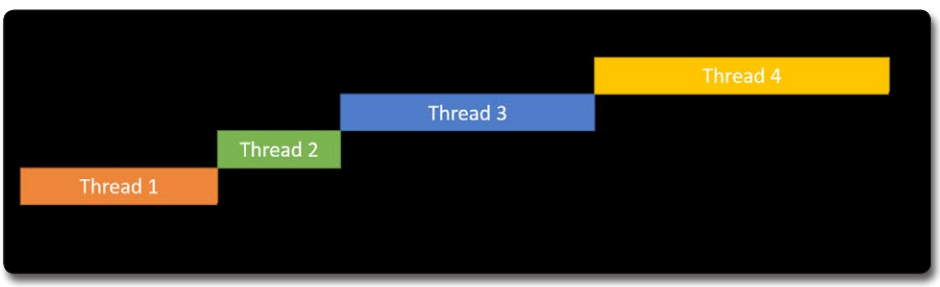

Figura 4. Escalonamento de threads, modo round-robin

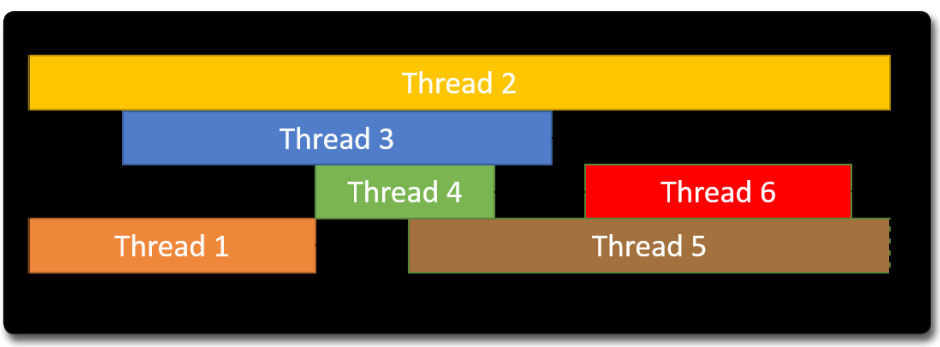

Figura 5. Escalonamento de threads no modo round-robin implementando paralelismo

- Estender a classe **Thread** (**java.lang.Thread**); e
- Implementar a interface **Runnable** (**java.lang.Runnable**).

Na **Listagem 1**, de forma simples e objetiva, é apresentado um exemplo de como implementar uma **Thread** para executar uma subtarefa em paralelo. Para isso, primeiramente é necessário codificar um **Runnable**, o que pode ser feito diretamente na criação da Thread, como demonstrado na **Listagem 1**, ou implementar uma classe própria que estenda **Runnable**. Posteriormente, basta executá-lo com um objeto **Thread** através do método **start()**.

Neste exemplo pode-se observar também o código utilizado para buscar alguns dados da thread atual, tais como ID, nome, prioridade, estado e até mesmo capturar o código que ela está executando. Além de tais informações que podem ser capturadas, é possível manipular as threads utilizando alguns dos seguintes métodos:

• O método estático **Thread.sleep()**, por exemplo, faz com que a thread em execução espere por um período de tempo sem consumir muito (ou possivelmente nenhum) tempo de CPU;
• O método **join()** congela a execução da thread corrente e aguarda a conclusão da thread na qual esse método foi invocado;
• Já o método **wait()** faz a thread aguardar até que outra invoque o método **notify()** ou **notifyAll()**; e
• O método **interrupt()** acorda uma thread que está dormindo devido a uma operação de **sleep()** ou **wait()**, ou foi bloqueada por causa de um processamento longo de I/O.

Listagem 1. Exemplo de thread implementando a interface Runnable.

```
public class ExemploThread {

   public static void main(String[] args) {
      new Thread(new Runnable() {
         @Override
         public void run() {
            //código para executar em paralelo
            System.out.println("ID: " + Thread.currentThread().getId());
            System.out.println("Nome: " + Thread.currentThread().getName());
            System.out.println("Prioridade: " + Thread.currentThread().getPriority());
            System.out.println("Estado: " + Thread.currentThread().getState());
         }
      }).start();
   }
}
```

A forma clássica de se criar uma thread é estendendo a classe **Thread**, como demonstrado na **Listagem 2**. Neste código, temos a classe **Tarefa** estendendo a **Thread**. A partir disso, basta sobrescrever o método **run()**, o qual fica encarregado de executar o código da thread.

Na prática, nossa classe **Tarefa** é responsável por realizar o somatório do intervalo de valores recebido no momento em que ela é criada e armazená-lo em uma variável para que possa ser lido posteriormente.

Para testarmos o paralelismo com a classe da **Listagem 2**, criamos a classe **Exemplo** com o método **main()**, responsável por executar o programa (vide **Listagem 3**). Neste exemplo, após criar as threads, chama-se o método **start()** de cada uma delas, para que iniciem suas tarefas. Logo após, em um bloco *try-catch*, temos a invocação dos métodos **join()**. Este faz com que o programa aguarde a finalização de cada thread para que depois possa ler o valor totalizado por cada tarefa.

Observe, na **Listagem 3**, que cada tarefa recebe seu intervalo de valores a calcular, sendo somado, ao todo, de 0 a 3000, mas e se tivéssemos uma única lista de valores que gostaríamos de somar para obter o valor total? Neste caso, as threads precisariam concorrer pela lista. Isso é o que chamamos de concorrência de dados e geralmente traz consigo diversos problemas.

Concorrência de dados

A concorrência de dados é um dos principais problemas a se enfrentar quando empregamos multithreading em uma aplicação.

Listagem 2. Código da classe Tarefa estendendo a classe Thread.

```
public class Tarefa extends Thread {

   private final long valorInicial;
   private final long valorFinal;
   private long total = 0;

   //método construtor que receberá os parâmetros da tarefa
   public Tarefa(int valorInicial, int valorFinal) {
      this.valorInicial = valorInicial;
      this.valorFinal = valorFinal;
   }

   //método que retorna o total calculado
   public long getTotal() {
      return total;
   }

   /*
    Este método se faz necessário para que possamos dar start() na Thread
    e iniciar a tarefa em paralelo
    */
   @Override
   public void run() {
      for (long i = valorInicial; i <= valorFinal; i++) {
         total += i;
      }
   }
}
```

Listagem 3. Código da classe Exemplo, utiliza a classe Tarefa.

```
public class Exemplo {

   public static void main(String[] args) {
      //cria três tarefas
      Tarefa t1 = new Tarefa(0, 1000);
      t1.setName("Tarefa1");
      Tarefa t2 = new Tarefa(1001, 2000);
      t2.setName("Tarefa2");
      Tarefa t3 = new Tarefa(2001, 3000);
      t3.setName("Tarefa3");

      //inicia a execução paralela das três tarefas, iniciando três novas threads no
      //programa
      t1.start();
      t2.start();
      t3.start();

      //aguarda a finalização das tarefas
      try {
         t1.join();
         t2.join();
         t3.join();
      } catch (InterruptedException ex) {
         ex.printStackTrace();
      }

      //Exibimos o somatório dos totalizadores de cada Thread
      System.out.println("Total: " + (t1.getTotal() + t2.getTotal() + t3.getTotal()));
   }
}
```

Ela é capaz de gerar desde inconsistência nos dados compartilhados até erros em tempo de execução. No entanto, felizmente isto pode ser evitado, sendo necessário, portanto, se precaver para que nosso aplicativo não apresente tais problemas.

Uma boa forma de evitar problemas de concorrência é sincronizar as threads que compartilham dados entre si. A partir disso, estas threads passam a executar em sincronia com outras, e assim,

uma por vez acessará o recurso. O sincronismo previne que duas ou mais threads acessem o mesmo recurso simultaneamente. Por outro lado, temos as threads assíncronas, que executam independentemente umas das outras e geralmente não compartilham recursos, como é o caso do exemplo das **Listagens 2** e **3**.

No exemplo da **Listagem 4**, por sua vez, é possível visualizar três threads disputando a mesma variável **varCompartilhada** para incrementá-la de forma assíncrona. Basicamente, a ideia desse código é incrementar uma variável com diferentes valores e, a cada valor gerado, adicioná-lo em uma lista (**ArrayList**).

Listagem 4. Exemplo de concorrência utilizando lista assíncrona.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ExemploAssincrono1 {

   private static int varCompartilhada = 0;
   private static final Integer QUANTIDADE = 10000;
   private static final List<Integer> VALORES = new ArrayList<>();

   public static void main(String[] args) {

      Thread t1 = new Thread(new Runnable() {
         @Override
         public void run() {
            for (int i = 0; i < QUANTIDADE; i++) {
               VALORES.add(++varCompartilhada);
            }
         }
      });

      Thread t2 = new Thread(new Runnable() {
         @Override
         public void run() {
            for (int i = 0; i < QUANTIDADE; i++) {
               VALORES.add(++varCompartilhada);
            }
         }
      });

      Thread t3 = new Thread(new Runnable() {
         @Override
         public void run() {
            for (int i = 0; i < QUANTIDADE; i++) {
               VALORES.add(++varCompartilhada);
            }
         }
      });

      t1.start();
      t2.start();
      t3.start();

      try {
         t1.join();
         t2.join();
         t3.join();
      } catch (InterruptedException ex) {
         ex.printStackTrace();
      }

      int soma = 0;
      for (Integer valor : VALORES) {
         soma += valor;
      }
      System.out.println("Soma: " + soma);
   }
}
```

No entanto, ao executar este algoritmo é provável que seja gerada a exceção **java.lang.ArrayIndexOutOfBoundsException**, devido à concorrência pela lista, visto que há mais de uma thread tentando inserir dados nela. Como o "ponto fraco" desta estrutura de dados é seu mecanismo dinâmico de tamanho variável, a cada novo valor a ser inserido é preciso expandir a lista. Desta forma, a thread perde tempo para fazer esta operação, aumentando assim a possibilidade de ser pausada pelo escalonador. Quando isto acontece e alguma outra thread tenta realizar a mesma operação de **add()**, a exceção é gerada. Com o intuito de solucionar esse problema, uma das opções é adotar uma lista sincronizada, conforme o código a seguir:

```
private static final List<Integer> VALORES = Collections.synchronizedList
(new ArrayList<>());
```

Apesar de solucionar o problema anterior, ainda é possível que a thread sofra interrupção durante o incremento da variável **varCompartilhada** e passe a gerar valores inconsistentes. Isto porque no processo atual de incremento da variável, primeiramente deve ser pego o valor atual desta, somá-lo com 1 e então obter o novo valor a ser armazenado.

Esse problema acontece porque nesse código existem três threads alterando o valor da mesma variável (nesse caso, com o operador **++**) e o escalonador, quando aloca uma thread ao processador, permite que ela execute seu código por um determinado período de tempo e depois a interrompe, possibilitando que outra thread ocupe seu lugar e opere sobre os mesmos dados. Assim, quando a thread anterior voltar a processar, trabalhará com valores desatualizados.

Para aferir o resultado deste algoritmo, toda atualização de valor da variável **varCompartilhada** é adicionada a uma lista e ao final é realizada a soma de todos esses valores. Por causa das situações supracitadas, no entanto, o resultado gerado a cada execução pode ser diferente. Isto demonstra que o incremento de uma variável assíncrona em threads é, sem dúvidas, um problema.

Nota

É preciso destacar que nem sempre ocorrerá esse problema, ou seja, nem sempre uma thread será interrompida durante o seu processamento. Para aumentar as chances desse problema acontecer, foi utilizado um intervalo de 10.000 repetições e três threads. Com um número baixo de iterações, coincidentemente pode ser gerado o mesmo resultado em quase todas as execuções.

O exemplo apresentado na **Listagem 5** traz uma derivação do código da **Listagem 4**. Neste caso, o **List** foi substituído por um **Set**, que suporta a inserção de valores de modo assíncrono e ainda garante a unicidade dos valores inseridos. Assim, não mais teremos problemas com o **ArrayList** e poderemos dar sequência à demonstração do problema de concorrência com a **varCompartilhada**.

Ao executar este algoritmo diversas vezes é possível observar (vide **Figura 6**) que ele imprime no console alguns valores a serem inseridos que já existem no **Set**, o que demonstra que as

threads estão incrementando a variável, mas em algum momento geram o mesmo valor. Isso acontece por causa da concorrência pela variável **varCompartilhada** de maneira assíncrona, onde ao incrementar esta variável, mais de uma thread acaba gerando o mesmo valor.

```
run:
Já existe: 1120
Já existe: 1672
Já existe: 1845
Já existe: 2286
Já existe: 2751
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 6. Resultado no console com a execução da **Listagem 5**

Listagem 5. Exemplo de concorrência utilizando HashSet.

```
import java.util.HashSet;
import java.util.Set;

public class ExemploAssincrono2 {

  private static int varCompartilhada = 0;
  private static final Integer QUANTIDADE = 10000;
  private static final Set<Integer> VALORES = new HashSet<>();

  public static void main(String[] args) {

    new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          boolean novo = VALORES.add(++varCompartilhada);
          if (!novo) {
            System.out.println(“Já existe: ” + varCompartilhada);
          }
        }
      }
    }).start();
    new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          boolean novo = VALORES.add(++varCompartilhada);
          if (!novo) {
            System.out.println(“Já existe: ” + varCompartilhada);
          }
        }
      }
    }).start();
    new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          boolean novo = VALORES.add(++varCompartilhada);
          if (!novo) {
            System.out.println(“Já existe: ” + varCompartilhada);
          }
        }
      }
    }).start();
  }
}
```

Sincronização de Threads

Caso não seja uma opção substituir o **ArrayList**, uma alternativa para solucionar o problema obtido na **Listagem 4** é sincronizar o objeto concorrido; neste caso, a lista (vide **Listagem 6**). Isso é possível porque todo objeto Java possui um lock associado, que pode ser disputado por qualquer trecho de código sincronizado e em qualquer thread.

Um bloco sincronizado previne que mais de uma thread consiga executá-lo simultaneamente. Para isso, a thread que for utilizar esse bloco adquire o lock associado ao objeto sincronizado e as demais que tentarem acessá-lo entrarão em estado de BLOCKED, até que o objeto seja liberado. Na **Figura 7** é possível observar o ciclo de vida de uma thread, da sua criação à sua finalização.

Listagem 6. Exemplo de sincronização de variável com bloco de código sincronizado.

```
import java.util.ArrayList;
import java.util.List;

public class ExemploBlocoSincronizado {

  //declaração das variáveis - vide Listagem 4

  public static void main(String[] args) {

    Thread t1 = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          synchronized (VALORES) {
            VALORES.add(++varCompartilhada);
          }
        }
      }
    });

    Thread t2 = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          synchronized (VALORES) {
            VALORES.add(++varCompartilhada);
          }
        }
      }
    });

    Thread t3 = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          synchronized (VALORES) {
            VALORES.add(++varCompartilhada);
          }
        }
      }
    });

    //Idem Listagem 4...
  }

}
```

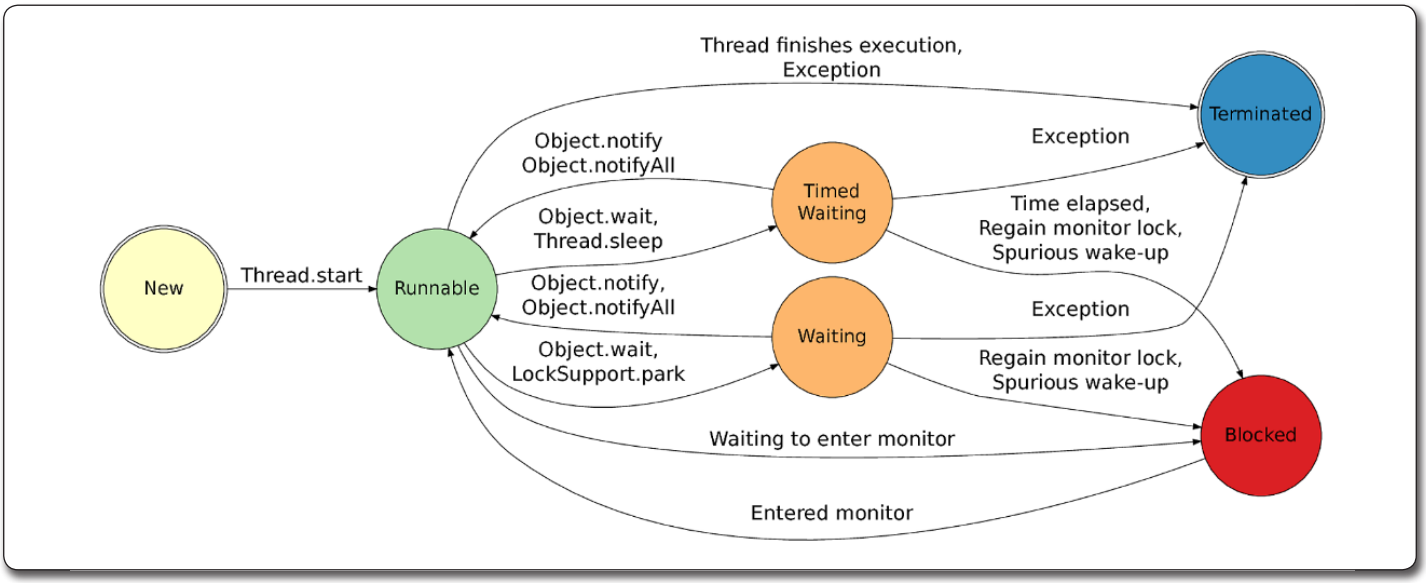

Figura 7. Ciclo de vida de uma thread

A seguir são descritos os possíveis estados que elas podem assumir:

- **New**: A thread dica neste estado após criar sua instância e antes de invocar o método **start()**;
- **Runnable:** Indica que ela está executando na máquina virtual Java;
- **Blocked:** Ainda está ativa, mas está à espera por algum recurso que está em uso por outra thread;
- **Waiting:** Quando neste estado, ela está à espera por tempo indeterminado pelo fato de outra thread ter executado uma determinada ação. Isto ocorre quando se invoca o método **wait()** ou **join()**, por exemplo;
- **Timed_Waiting:** Neste estado a thread está à espera de uma operação por um tempo pré-determinado. Por exemplo, esta situação ocorre ao invocar métodos como **Thread.sleep(sleeptime)**, **wait(timeout)** ou **join(timeout)**; e
- **Terminated:** Este estado sinaliza que o método **run()** finalizou.

> **Nota**
>
> Ao sincronizar operações, prefira sempre o uso de métodos sincronizados no lugar de blocos desse tipo. Isso porque os bytecodes gerados para um método sincronizado são relativamente menores do que os gerados para um bloco sincronizado.

Outra forma de acessar um dado compartilhado entre threads é criando um método sincronizado. Essa técnica é muito parecida com a anterior, mas ao invés de sincronizar o mesmo bloco de código em cada thread, ele é transferido para um método que contém a notação **synchronized** na assinatura. Assim, as threads terão que invocá-lo para realizar a operação sobre o dado concorrente. Veja um exemplo na **Listagem 7**.

Variáveis atômicas

Quando é preciso utilizar tipos primitivos de forma concorrente uma boa opção é adotar seu respectivo tipo atômico,

Listagem 7. Exemplo de método sincronizado.

```
import java.util.ArrayList;
import java.util.List;

public class ExemploMetodoSincronizado {

  //Idem Listagem 4...

  public static void main(String[] args) {

    Thread t1 = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          incrementaEAdd();
        }
      }
    });

    Thread t2 = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          incrementaEAdd();
        }
      }
    });

    Thread t3 = new Thread(new Runnable() {
      @Override
      public void run() {
        for (int i = 0; i < QUANTIDADE; i++) {
          incrementaEAdd();
        }
      }
    });

//Idem Listagem 4
  }

  private synchronized static void incrementaEAdd() {
    VALORES.add(++varCompartilhada);
  }

}
```

presente no pacote **java.util.concurrent.atomic**. Este tipo de objeto disponibiliza operações como incremento através de métodos próprios e são executadas em baixo nível de hardware, de forma que a thread não será interrompida durante o processo. Deste modo não é necessário sincronizar o objeto, gerando um algoritmo sem bloqueios e muito mais rápido. Veja o código a seguir:

```
private static AtomicInteger varCompartilhada = new AtomicInteger(0);
```

Neste caso, ao invés de utilizar um **Integer** para armazenar o valor, foi instanciado um **AtomicInteger**. Com isso, pode-se trocar o **varCompartilhada++** pela chamada **varCompartilhada.incrementAndGet()**, que realizará uma função semelhante de forma atômica, o que garantirá que a thread não seja interrompida no meio do processo de incremento da variável.

Nota

O ato de adquirir bloqueios para sincronizar threads consome tempo, mesmo quando nenhuma precisa aguardar a liberação do objeto sincronizado. Esse processo é uma faca de dois gumes: se por um lado ele resolve problemas de concorrência, por outro serializa o processamento das threads sobre esse bloco; ou seja, as threads nunca estarão processando esse código simultaneamente, o que pode degradar o desempenho. Portanto, esse recurso deve ser usado com moderação e somente onde for necessário.

Nota

Em tipos atômicos, métodos que não modificam seu valor são sincronizados.

Interface Callable

A interface **Runnable** é utilizada desde as primeiras versões da plataforma Java e como todos já sabem, ela fornece um único método – **run()** – que não aceita parâmetros e não retorna valor, assim como não pode lançar qualquer tipo de exceção. No entanto, e se precisássemos executar uma tarefa em paralelo e ao final obter um resultado como retorno? Para solucionar esse problema, você poderia criar um método na classe que implementa **Thread** ou **Runnable** e esperar pela conclusão da tarefa para acessar o resultado, assim como no cenário da **Listagem 8**.

Basicamente não há nada de errado com esse código, mas a partir do Java 5 este processo pode ser feito de forma diferente, graças à interface **Callable**. Deste modo, em vez de ter um método **run()**, a interface **Callable** oferece um método **call()**, que pode retornar um objeto qualquer, além da grande vantagem de poder capturar uma exceção gerada pela tarefa da thread.

Listagem 8. Exemplo de leitura de resultado em tarefa com Thread.

```
ThreadTarefa t = new ThreadTarefa();
t.start(); //inicia o trabalho da thread
t.join();  //aguarda a thread finalizar
String valor = t.getRetornoTarefa();
//acessa o resultado do processamento da tarefa.
```

Para tirar proveito dos benefícios de um objeto **Callable**, é altamente recomendável não utilizar um objeto **Thread** para executá-lo, e sim alguma outra API, como:

- **ExecutorService**: É uma API de alto nível para trabalhar diretamente com threads. Permite criar um pool de threads, reutilizá-las e gerenciá-las; e
- **ExecutorCompletionService:** É uma implementação da interface **CompletionService** que, associada a um **ExecutorService**, permite, através do método **take()**, receber o resultado de cada tarefa conforme elas vão finalizando, independente da ordem em que as tarefas foram criadas.

As implementações apresentadas nas **Listagens 9** e **10** demonstram uma boa prática no uso de Callables. Este código cria três tarefas que levam um determinado tempo para concluir e, ao terminar, retornam o nome da thread que a realizou. O código da tarefa se encontra na classe **ExemploCallable**, que implementa a interface **Callable**, com retorno do tipo **String**. Com esta interface a tarefa que se deseja executar deve ser implementada no método **call()** (vide **Listagem 9**), o qual é invocado ao executar o objeto **Callable**.

O código da **Listagem 10** tem o objetivo de criar e executar três tarefas armazenadas em uma lista. Para simular uma diferença no tempo de execução das threads, cada uma foi desenvolvida para aguardar um certo tempo em milissegundos, que lhe é fornecido no método construtor. Antes de executá-las, no entanto, note que é criado um pool de threads com um **ExecutorService**, o qual posteriormente é utilizado para criar um **ExecutorCompletionService**, que será encarregado de executar as tarefas e também nos será útil para receber o retorno de cada uma delas conforme forem concluindo.

Dito isso, uma a uma as tarefas são executadas através do método **submit()** e, por fim, é utilizado o método **take()**, para buscar a tarefa concluída, e o método **get()**, que lê o retorno dela e o imprime no console (**Figura 8**).

```
run:
Tarefas iniciadas, aguardando conclusão
pool-1-thread-2
pool-1-thread-3
pool-1-thread-1
CONSTRUÍDO COM SUCESSO (tempo total: 8 segundos)
```

Figura 8. Resultado da **Listagem 10** no console

Nota

Note, pelo resultado da **Figura 8**, que, por mais que a tarefa que tinha duração de oito segundos seja executada primeiro, seu resultado aparece por último. Esse resultado é obtido porque a leitura dos retornos de cada tarefa não tem relação com a ordem de execução, e sim com sua conclusão, graças ao ExecutorCompletionService.

Listagem 9. Exemplo de classe implementando Callable.

```
import java.util.concurrent.Callable;

public class ExemploCallable implements Callable<String> {

   private final long tempoDeEspera;

   public ExemploCallable(int time) {
      this.tempoDeEspera = time;
   }

   @Override
   public String call() throws Exception {
      Thread.sleep(tempoDeEspera);
      return Thread.currentThread().getName();
   }
}
```

Listagem 10. Exemplo de tarefas com retorno utilizando Callable.

```
package javamagazine.threads;

import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExemploRetornoDeTarefa {

   public static void main(String[] args) {
      List<ExemploCallable> tarefas = Arrays.asList(
            new ExemploCallable(8000),
            new ExemploCallable(4000),
            new ExemploCallable(6000));

      ExecutorService threadPool = Executors.newFixedThreadPool(3);
      ExecutorCompletionService<String> completionService =
      new ExecutorCompletionService<>(threadPool);

      //executa as tarefas
      for (ExemploCallable tarefa : tarefas) {
         completionService.submit(tarefa);
      }
      System.out.println("Tarefas iniciadas, aguardando conclusão");

      //aguarda e imprime o retorno de cada uma
      for (int i = 0; i < tarefas.size(); i++) {
         try {
            System.out.println(completionService.take().get());
         } catch (InterruptedException | ExecutionException ex) {
            ex.printStackTrace();
         }
      }

      threadPool.shutdown();
   }
}
```

Coleções concorrentes vs Coleções sincronizadas

Um recurso bastante utilizado no desenvolvimento de software são as coleções de dados. Na plataforma Java estas estruturas estão disponíveis em uma série de implementações para os mais diversos fins. Como sabemos, não há nenhum "mistério" em declará-las, no entanto, como é comum nos depararmos com bugs ao acessar essas estruturas de maneira concorrente, vale dedicar um tópico deste artigo para explorar suas peculiaridades.

Dentre as coleções disponíveis no Java, existem variados tipos de estruturas de dados, como, listas, pilhas e filas, e estas, por sua vez, ainda se subdividem quanto a forma de implementação, que compreende:

- **Coleções sem suporte a threads:** São as coleções normalmente utilizadas. Encontradas no pacote **java.util**, como **ArrayList**, **HashMap**, **HashSet**, não devem ser utilizadas de forma concorrente, a menos que seja feito um sincronismo externo sobre a coleção;
- **Coleções sincronizadas:** Podem ser criadas a partir de métodos estáticos disponíveis na classe **java.util.Collections**, por exemplo: **java.util.Collections.synchronizedList(objetoLista)**. Como estes métodos retornam uma coleção sincronizada, isto significa que seu acesso para modificações ocorre de forma serializada, ou seja, somente uma thread por vez pode acessá-la; e
- **Coleções concorrentes:** Não necessitam de nenhum sincronismo adicional, como sincronizar seu objeto ou algum método, pois possuem um sofisticado suporte para concorrência. Estas coleções, livres de problemas advindos da concorrência entre threads, podem ser encontradas no pacote **java.util .concurrent**.

Sabendo disso, preferencialmente, opte por utilizar coleções concorrentes, ao invés das sincronizadas, pois as coleções concorrentes possuem maior escalabilidade e suportam modificações simultâneas de diversas threads sem precisar estabelecer um bloqueio. Já as coleções sincronizadas têm sua performance degradada devido ao bloqueio que precisam estabelecer quando uma thread as acessa. Logo, isso também significa que somente uma thread por vez pode modificá-las.

Um detalhe que costuma passar despercebido nas entrelinhas da programação concorrente é que não existe a garantia de execução paralela ou de que cada thread vai executar em um núcleo diferente. Criar threads apenas sugere à JVM que aquilo seja paralelizado. Por exemplo, você pode ter um processador de quatro núcleos e criar um aplicativo com quatro threads que processem exaustivamente, mas isso não lhe garante que cada uma das quatro threads serão executadas por um núcleo diferente, tão pouco consumirão 100% de processamento. Portanto, não basta criar threads pensando que isto é a solução dos seus problemas. Neste caso, ao criar threads em demasia estar-se-ia degradando a performance, já que a JVM gastaria muito tempo com o escalonamento delas, se comparado ao tempo total de processamento utilizado pelas threads.

Primeiramente, a aplicação deve ser inteligente o bastante para criar o número ideal de threads, ou seja, deve ser levada em consideração a quantidade de processadores/núcleos disponíveis no sistema. Criar um número de threads menor do que o número de núcleos disponíveis gera desperdício. Por outro lado, gerar um número excessivamente maior de threads, causará outro problema. Será perdido mais tempo com o escalonamento das threads do que com as próprias tarefas que elas precisam executar, e assim, por mais que se esteja consumindo 100% da CPU, não se tem o desempenho máximo que se pode atingir.

Para amenizar este problema, um recurso muito útil da plataforma Java pode ser verificado no código apresentado a seguir, que permite ler a quantidade de núcleos disponíveis. A partir disso, podemos calcular o número ideal de threads necessárias para atingir os 100% de processamento sem desperdícios, quando temos uma aplicação que precisa realizar um cálculo exaustivo:

```
int nucleos = Runtime.getRuntime().availableProcessors();
```

Framework Fork/Join

O framework Fork/Join, introduzido na versão 7 da plataforma Java, é uma implementação da interface **ExecutorService** que auxilia o desenvolvedor a tirar proveito das arquiteturas multicore. Esta API foi projetada para as tarefas que podem ser quebradas em pequenas partes recursivamente, com o objetivo de usar todo o poder de processamento disponível para melhorar o desempenho da aplicação.

O exemplo apresentado nas **Listagens 11** e **12** demonstra um cenário onde o objetivo é buscar, recursivamente em um sistema de arquivos, os arquivos com determinada extensão. Ao iniciar, a tarefa recebe um diretório base onde o algoritmo começa as buscas. O conteúdo do diretório é então analisado e caso haja outra pasta dentro desta, é criada outra tarefa para analisar aquele diretório, e assim recursivamente o algoritmo realiza a busca pelos arquivos e retorna os resultados à tarefa pai.

Tecnicamente, para realizar este processo foi implementada uma classe que estende **RecursiveTask** e recebe um **List** de **String**, o qual é utilizado para informar o tipo de retorno da tarefa (vide **Listagem 11**). Ao criar a tarefa, ou seja, uma instância da classe **ProcessadorDePastas**, é necessário informar por parâmetros o diretório base onde se iniciará a busca e a extensão de arquivo pela qual se dará a busca.

Quando se estende a classe **RecursiveTask**, deve ser implementado o método **compute()**, que é responsável por desempenhar a tarefa desejada, assim como devemos codificar o método **run()**, quando se implementa a interface **Runnable**. É neste método que está especificada a busca pelos arquivos. Nele, o ponto mais importante pode ser verificado na recursividade, local que cria as tarefas paralelas com a chamada ao método **fork()** para cada pasta localizada dentro da pasta na qual se está pesquisando. Ao final, cada subtarefa retorna os dados de sua busca à tarefa que a criou, e esta, por sua vez, adiciona estes dados na lista "**tarefas**". Este é o processo de desempilhar a recursão, que é realizado até chegar à primeira tarefa criada na classe **ForkJoinMain**, momento este em que os dados são retornados para a lista **resultados** pelo método **join()** (vide **Listagem 12**).

Na **Listagem 12** temos o código responsável por iniciar a tarefa principal, ler e exibir os resultados. Para tal, foram criadas três tarefas base que farão as buscas em três pastas distintas, e a fim de

Listagem 11. Exemplo de tarefa Fork/Join.

```
import java.io.File;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class ProcessadorDePastas extends RecursiveTask<List<String>> {

  private final String diretorio;
  private final String extensao;

  public ProcessadorDePastas(String diretorio, String extension) {
    this.diretorio = diretorio;
    this.extensao = extension;
  }

  @Override
  protected List<String> compute() {
    List<String> lista = new ArrayList<>();
    List<ProcessadorDePastas> tarefas = new ArrayList<>();
    File arquivo = new File(diretorio);
    File conteudo[] = arquivo.listFiles();

    if (conteudo != null) {
      for (int i = 0; i < conteudo.length; i++) {
        if (conteudo[i].isDirectory()) {
          ProcessadorDePastas tarefa = new ProcessadorDePastas(conteudo[i].
          getAbsolutePath(), extensao);
          tarefa.fork();
          tarefas.add(tarefa);
        } else if (verificaArquivo(conteudo[i].getName())) {
          lista.add(conteudo[i].getAbsolutePath());
        }
      }
    }
    if (tarefas.size() > 50) {
      System.out.printf("%s: %d tarefas executando.\n",
      arquivo.getAbsolutePath(), tarefas.size());
    }
    addResultadosDaTarefa(lista, tarefas);
    return lista;
  }

  private void addResultadosDaTarefa(List<String> lista,
  List<ProcessadorDePastas> tarefas) {
    for (ProcessadorDePastas item : tarefas) {
      lista.addAll(item.join());
    }
  }

  private boolean verificaArquivo(String nome) {
    return nome.endsWith(extensao);
  }
}
```

executá-las, foi instanciado um pool de threads com um **ForkJoinPool**. Este tipo de pool gerencia de forma mais eficiente o trabalho das threads, pois utiliza uma técnica chamada de "roubo de tarefa" para executar as tarefas em espera. Nesta abordagem cada thread possui uma fila de tarefas em espera e no momento em que uma thread não tiver mais nada em sua fila, poderá "roubar" o trabalho de outra, possibilitando mais uma melhoria na performance.

Por fim, saiba que enquanto o aplicativo processa é possível extrair algumas informações úteis, a fim de monitorar o trabalho do framework e do pool de threads. Estes dados podem ser obtidos com o próprio objeto do pool, através dos seguintes métodos:

- **getParallelism():** Retorna o nível do paralelismo. Por default e por recomendação, é a quantidade de núcleos do processador;
- **getActiveThreadCount():** Retorna a quantidade de threads ativas;
- **getQueuedTaskCount():** Retorna a quantidade total de tarefas na fila de espera; e
- **getStealCount():** Retorna a quantidade de roubos que ocorreram. Um roubo ocorre quando uma thread fica sem trabalho. Então ela rouba tarefas da fila de espera de outra thread.

Listagem 12. Exemplo de uso da tarefa Fork/Join.

```
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.TimeUnit;

public class ForkJoinMain {

   public static void main(String[] args) {
      ProcessadorDePastas sistema = new ProcessadorDePastas("C:/Windows", ".exe");
      ProcessadorDePastas aplicativos = new ProcessadorDePastas
      ("C:/Program Files", ".exe");
      ProcessadorDePastas documentos = new ProcessadorDePastas("C:/users", ".doc");

      ForkJoinPool pool = new ForkJoinPool();
      pool.execute(sistema);
      pool.execute(aplicativos);
      pool.execute(documentos);

      do {
         System.out.printf("------------------------------------------\n");
         System.out.printf("-> Paralelismo: %d\n", pool.getParallelism());
         System.out.printf("-> Threads Ativas: %d\n", pool.getActiveThreadCount());
         System.out.printf("-> Tarefas: %d\n", pool.getQueuedTaskCount());
         System.out.printf("-> Roubos: %d\n", pool.getStealCount());
         System.out.printf("------------------------------------------\n");
         try {
            TimeUnit.SECONDS.sleep(1);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      } while ((!sistema.isDone()) || (!aplicativos.isDone()) || (!documentos.isDone()));

      pool.shutdown();

      List<String> resultados;
      resultados = sistema.join();
      System.out.printf("Sistema: %d aplicativos encontrados.\n", resultados.size());
      resultados = aplicativos.join();
      System.out.printf("Aplicativos: %d encontrados.\n", resultados.size());
      resultados = documentos.join();
      System.out.printf("Documentos: %d  encontrados.\n", resultados.size());
   }
}
```

Java 8 – Lambdas e Streams

A versão 8 da plataforma Java tem como uma das suas principais características o suporte a expressões Lambda, recurso que foi projetado com o intuito de facilitar a programação funcional e reduzir o tempo de desenvolvimento. Isso pode ser exemplificado criando um objeto **Thread**, como expõe o código da **Listagem 13**, onde é possível notar que a criação do objeto **Runnable** se torna implícita, reduzindo de cinco para duas a quantidade de linhas necessárias para a criação de uma thread.

Ainda no Java 8, uma nova abstração, chamada **Stream**, foi desenvolvida. Esta permite processar dados de forma declarativa, assim como possibilita a execução de tarefas utilizando vários núcleos sem que seja necessário implementar uma linha de código multithreading, através da função **parallelStream**. Quando um stream é executado em paralelo, a JVM o particiona em vários substreams, os quais são iterados individualmente por threads e, por fim, seus resultados são combinados (veja **Listagem 14**).

Além de ser extremamente simples e funcional, em poucas linhas é possível extrair várias informações de uma lista numérica, como valor máximo, mínimo, soma total e média, sem ter que se preocupar em desenvolver estas funções.

Listagem 13. Criando uma thread com expressões lambda.

```
new Thread(() -> {
//Código da tarefa a ser executada
}).start();
```

Listagem 14. Exemplo utilizando ParallelStream.

```
import java.util.ArrayList;
import java.util.List;
import java.util.LongSummaryStatistics;
import java.util.Random;

public class ExemploParallelStream {

   public static void main(String[] args) {
      List<Long> numeros = new ArrayList<>();

      Random random = new Random();
      for (int i = 0; i < 10000000; i++) {
         numeros.add(random.nextLong());
      }

      LongSummaryStatistics stats = numeros.parallelStream().mapToLong((x) ->
      x ).summaryStatistics();
      System.out.println("Maior número na lista: " + stats.getMax());
      System.out.println("Menor número na lista: " + stats.getMin());
      System.out.println("Soma de todos os números: " + stats.getSum());
      System.out.println("Média de todos os números: " + stats.getAverage());
   }
}
```

E mesmo que sua lista não seja numérica, ainda assim se tornou mais fácil transformar ou extrair informações por meio das expressões lambda.

O Java foi uma das primeiras plataformas a fornecer suporte a multithreading no nível de linguagem e agora é uma das primeiras a padronizar utilitários e APIs de alto nível para lidar com threads, como a introdução do framework Fork/Join na versão 7, e a API de streams e o suporte a expressões lambda na versão 8.

Atualmente, qualquer computador ou smartphone tem mais de um núcleo de processamento e a cada novo lançamento esta quantidade só aumenta, assim como a importância do software ser desenvolvido em multithreading. Atendendo a esse cenário, o Java fornece uma base sólida para a criação de uma ampla variedade de soluções paralelas.

Para finalizar, note que é possível alcançar bons resultados com as técnicas aqui demonstradas. Contudo, sempre utilize a programação concorrente com bastante atenção, pois ao manipular dados compartilhados entre threads poderá cair em alguns cenários de depuração bem difíceis.

Autor

Rodrigo Schieck
rodrigo.void@gmail.com
É bacharel em Ciência da Computação pela UNIJUI-RS. Trabalha no desenvolvimento de aplicações Java há seis anos, sendo três deles como professor de programação Java.

Desenvolvimento de aplicações corporativas com Apache TomEE

Aprenda a utilizar os recursos da plataforma Java EE disponibilizados pelo servidor de aplicações corporativas mais leve do mercado

Este artigo é do tipo mentoring

saiba mais: www.devmedia.com.br/mentoring-saibamais

Embora exista no mercado servidores como WildFly, GlassFish, WebSphere, entre outros, o Apache Tomcat tem sido a alternativa mais popular e viável para a maioria dos desenvolvedores de aplicações Java. Apesar disso, apenas as tecnologias JSP e Servlet, implementadas pelo Tomcat, não atendem, em sua maioria, às demandas das soluções modernas. Com a evolução da plataforma Java EE, observa-se que há um grande abismo entre os recursos oferecidos pelo Tomcat e os necessários para a implementação de aplicações mais robustas, que envolvam vários componentes.

É claro que, ao criar uma aplicação corporativa é possível adicionar bibliotecas aos projetos e assim ampliar as possibilidades de desenvolvimento. No entanto, nesses casos fica a cargo do desenvolvedor identificar quais bibliotecas precisam ser adicionadas, assegurando que elas funcionarão corretamente e que não haverá conflito entre versões. O Apache TomEE, por ser um servidor Java EE, traz consigo implementações de várias APIs que ajudam a preencher esta lacuna.

No artigo "Conhecendo o Apache Tomcat e o TomEE", publicado na Java Magazine 146 (veja a seção **Links**), falamos sobre a origem deste servidor de aplicações que, baseado no Tomcat e sob o guarda-chuva da Apache Software Foundation, se apresenta como uma alternativa promissora aos servidores de aplicações pré-estabelecidos.

Neste artigo, apresentaremos os recursos que demonstram o poder do Apache TomEE. Para isso, faremos uma breve consideração sobre as versões disponíveis para download, falaremos sobre a configuração do ambiente de desenvolvimento e em seguida iniciaremos a implementação da aplicação utilizando os recursos Java EE de forma que seja possível rodá-la no TomEE. Após a leitura e a execução do exemplo, o leitor verá como é fácil e prático construir aplicações corporativas com esse servidor.

Cenário

O desenvolvimento de aplicações corporativas é uma atividade complexa que requer cuidados com transações, portabilidade, interoperabilidade, segurança, entre outros aspectos. Assim, a utilização de um servidor Java EE pode simplificar consideravelmente essas preocupações, deixando aos cuidados do servidor o gerenciamento destes recursos. Ciente disso e visando apresentar uma alternativa aos desenvolvedores e arquitetos, a Apache lançou, em abril de 2012, o Apache TomEE, um servidor leve e robusto, baseado no Apache Tomcat, que implementa as especificações da plataforma Java EE.

Considerando essa opção, este artigo apresenta os recursos desse servidor de aplicações e demonstra, através de um exemplo prático, os ganhos que ele traz para aqueles que precisam desenvolver aplicações corporativas, mas querem evitar as preocupações com bibliotecas e suas dependências. Ao adotá-lo, os esforços serão concentrados nos requisitos funcionais, agilizando e simplificando o processo de desenvolvimento.

Download do Apache TomEE

Antes de iniciarmos, é importante lembrar que o Apache TomEE está disponível em três distribuições: Perfil Web (Web Profile), JAX-RS e Plus. A primeira, compatível com o Perfil Web da especificação Java EE 6; a segunda, também construída com base no Perfil Web, mas contando com a adição do suporte a JAX-RS, através de uma versão simplificada do Apache CXF; e a distribuição Plus, que fornece um pacote completo com todos os componentes disponíveis para TomEE, incluindo JMS, JAX-WS e JCA. Ambas as distribuições estão disponíveis na versão 1.7.3 e rodam sobre o Apache Tomcat 7.x.

Além destas, já está disponível o TomEE 7.0 Milestone 1, que roda sobre o Tomcat 8.x e encontra-se na fase de testes. No exemplo que estamos desenvolvendo trabalharemos com a distribuição PluME versão 1.7.3, que utiliza as implementações Mojarra, para JSF, e Eclipse Link, para JPA (ver **Tabela 1**). O site para download do Apache TomEE está disponível na seção **Links**.

Para rodar o Apache TomEE é necessário instalar o Java Development Kit e configurar a variável de ambiente *JAVA_HOME*. Embora não seja necessário para o nosso exemplo, também é interessante adicionar a variável *CATALINA_HOME* apontando para a pasta principal do TomEE, que neste exemplo refere-se à *c:\apache-tomee-plume-1.7.3*.

As distribuições do TomEE, disponíveis no site, estão compactadas no formato zip ou gzip. Após o download e a descompactação do servidor, é possível executá-lo no ambiente Windows através da linha de comando ou com um clique duplo no arquivo *startup.bat*. Considerando que os arquivos tenham sido descompactados na raiz do disco C, assim como no Apache Tomcat, o TomEE poderá ser iniciado/encerrado com os comandos da **Listagem 1**. Após a inicialização, ao acessar através de um navegador a página *http://localhost:8080/*, será exibida uma tela indicando que ele está instalado e funcionando corretamente (ver **Figura 1**).

Componente	JSR	Tomcat	PluME	Implementação
Servlets 3.x	315	✓	✓	Apache Tomcat
JavaServer Pages (JSP)	245	✓	✓	Apache Tomcat
Java Standard Tag Library (JSTL)	52	✓	✓	Apache Tomcat
JavaServer Faces (JSF)	314		✓	Oracle Mojarra
Java Transaction API (JTA)	907		✓	Apache Geronimo
Java Persistence API (JPA)	317		✓	Eclipse Link
Java Context and Dependency Injection (CDI)	299		✓	Apache OpenWeb-Beans
Java Authentication and Authorization Service (JAAS)	196		✓	Apache Geronimo
Java Authorization Contract for Containers (JACC)	115		✓	Apache Geronimo
JavaMail API	919		✓	Apache Geronimo
Bean Validation	303		✓	Apache BVal
Enterprise JavaBeans (EJB)	318		✓	Apache OpenEJB
Java API for RESTful Web Services (JAX-RS)	311		✓	Apache CXF
Java API for XML Web Services (JAX-WS)	224		✓	Apache CXF
Java EE Connector Architecture (JCA)	323		✓	Apache Geronimo
Java Messaging Service (JMS)	919		✓	Apache ActiveMQ

Tabela 1. Especificações Java EE 6 e implementações utilizadas pelo Apache TomEE PluME

Listagem 1. Comandos de inicialização e encerramento do Apache TomEE.

```
c:\apache-tomee-plume-1.7.3\bin>startup.bat
c:\apache-tomee-plume-1.7.3\bin>shutdown.bat
```

Configuração do Apache TomEE no ambiente de desenvolvimento Eclipse

Ao configurar servidores no Eclipse é possível escolher um dos *servers adapters* distribuídos com ele ou adicionar um. Na configuração do Apache TomEE utiliza-se o *adapter* do Tomcat, que já vem com a IDE. Para isso, com o Eclipse aberto e a perspectiva Java EE selecionada, basta clicar na aba *Servers*, iniciar a inclusão de um novo servidor, escolher o *adapter* Tomcat v7.0 Server e, na definição do diretório de instalação, selecionar a pasta onde está o TomEE (ver **Figura 2**).

Após incluir o servidor é importante alterar algumas configurações para que o Eclipse controle a execução do TomEE. Desse modo, dê um clique duplo no servidor adicionado e altere a opção *Server Locations* para *Use Tomcat Installation (takes control of Tomcat Installation)*.

Desenvolvendo uma aplicação corporativa

Para demonstrar os recursos e as possibilidades de uso do Apache TomEE, vamos desenvolver um pequeno projeto corporativo que chamaremos de Livraria EE. Ainda que o projeto seja modesto do ponto de vista funcional, permitindo apenas listar, incluir, alterar e excluir títulos de um banco de dados, ele é relativamente complexo do ponto de vista não-funcional.

Só para se ter uma ideia, o exemplo utiliza nove componentes da plataforma Java EE: Servlets, JavaServer Faces (JSF), Java Persistence API (JPA), Java Transaction API (JTA), Java Contexts and Dependency Injection (CDI), Bean Validation, Enterprise JavaBeans (EJB), Java API for RESTful Web Services (JAX-RS) e Java API for XML Web Services (JAX-WS).

Além dos componentes, a solução utiliza um banco de dados para gravar títulos, usuários e permissões da aplicação. Assim, caso ainda não tenha feito, realize o download do MySQL, instale o SGBD e execute o script que aparece na **Listagem 2**. Esse script criará o banco de dados com as tabelas *livro*, *users* e *user_roles* e incluirá uma pequena amostra de registros no banco.

Para que o TomEE consiga gerenciar as conexões com o banco de dados, o arquivo *mysql-connector-java-xxxx-bin.jar*, disponível no site do MySQL, precisa ser colocado na pasta *lib* do servidor TomEE. Ademais, as informações de conexão com o banco de dados devem ser configuradas no arquivo *tomee.xml* (ver **Listagem 3**) que, por sua vez, está na pasta *conf* do servidor. É importante assegurar que o usuário e a senha, informados neste arquivo de configuração, correspondam às credenciais de autenticação fornecidas durante a instalação do SGBD.

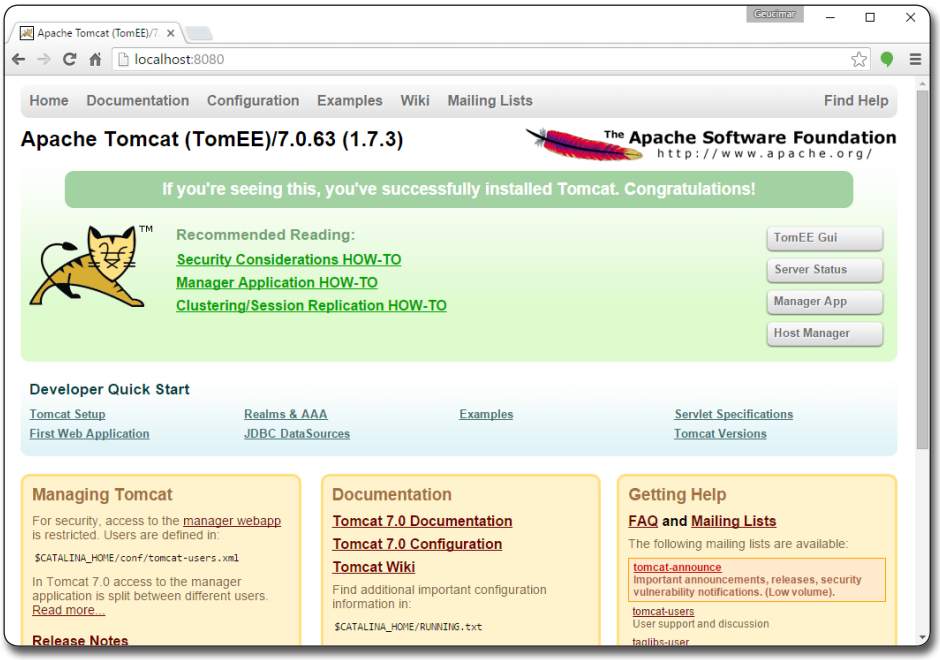

Figura 1. Tela inicial do servidor Apache TomEE

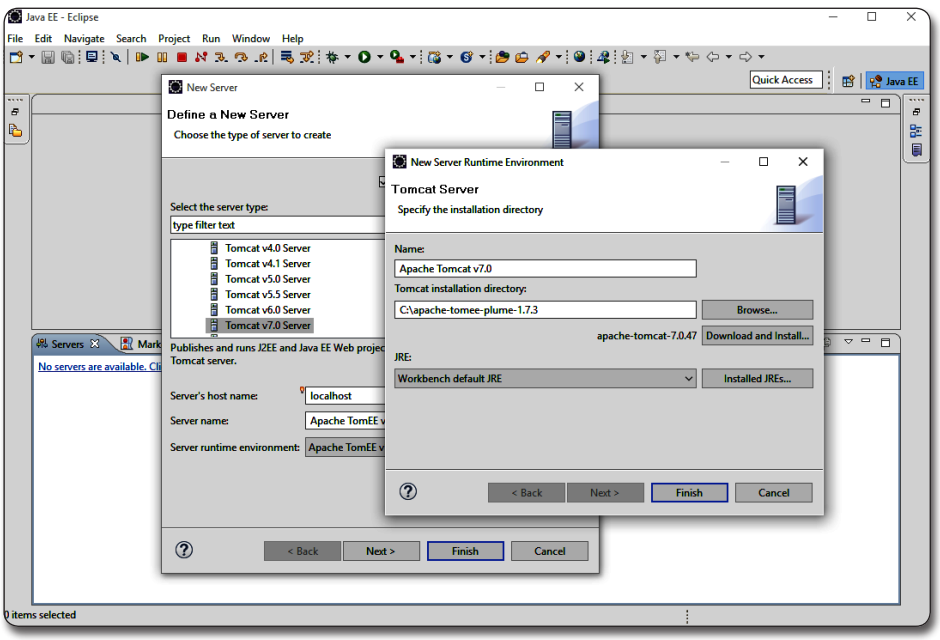

Figura 2. Telas da configuração do servidor Apache TomEE no Eclipse

Arquitetura corporativa baseada no modelo de distribuição Web Archive (WAR)

Até na plataforma Java EE 6, a forma de distribuição de aplicações corporativas era através da criação de um projeto *Enterprise ARchive (EAR)* e a vinculação dos demais módulos a ele. No entanto, a especificação JSR-316 retirou essa obrigatoriedade, possibilitando a distribuição em *Web ARchives* (WAR). Desde então, embora seja possível, também não é mais necessário criar EJBs em projetos separados. Essa forma de distribuição de aplicações corporativas em arquivos WAR é originária do TomEE e ficou conhecida como *Collapsed EAR in WAR*, sendo a primeira grande contribuição dele para a plataforma Java EE.

Na arquitetura da aplicação corporativa Livraria EE, vamos trabalhar com um projeto EJB em separado para obter baixo acoplamento e facilitar sua reutilização (ver **Figura 3**). Desta forma, a solução será constituída em três módulos: o módulo de serviços, **LivrariaEJB**, criado como um *EJB Project* no Eclipse; e os módulos **LivrariaWEB** e **LivrariaWS**, criados com a opção *Dynamic Web Project*.

O módulo LivrariaEJB, que contém os *Enterprise JavaBeans*, implementa as regras de negócio da aplicação e disponibiliza serviços que são consumidos diretamente pela aplicação LivrariaWEB. Estes mesmos serviços também são disponibilizados como web services SOAP/RESTful através da aplicação LivrariaWS e podem ser consumidos em arquiteturas orientadas a serviços (SOA) ou através de *front-ends* modernos como AngularJS, Backbone.js, Ember.js, etc.

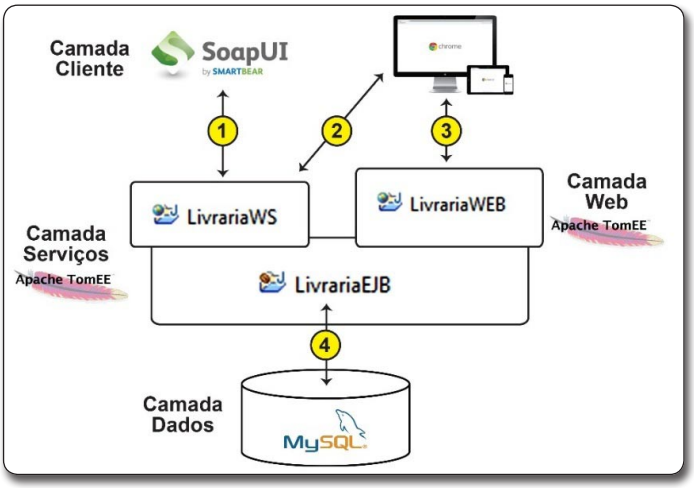

Figura 3. Arquitetura da Livraria EE

Listagem 2. Script para criação do banco de dados.

```
CREATE DATABASE `livraria` /*!40100 DEFAULT CHARACTER SET utf8 */;

CREATE TABLE `livraria`.`livro` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `nome` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
)  ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `livraria`.`users` (
  `user_name` varchar(255) NOT NULL,
  `user_pass` varchar(255) DEFAULT NULL,
k  PRIMARY KEY (`user_name`)
)  ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `livraria`.`user_roles` (
  `role_name` varchar(255) NOT NULL,
  `user_name` varchar(255) NOT NULL,
  PRIMARY KEY (`role_name`,`user_name`)
)  ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `livraria`.`livro` (`nome`) VALUES ('Real World Java EE Patterns
Rethinking Best Practices, Adam Bien');
INSERT INTO `livraria`.`livro` (`nome`) VALUES ('Patterns of Enterprise Application
Architecture, Martin Fowler');
INSERT INTO `livraria`.`livro` (`nome`) VALUES ('Enterprise Integration Patterns:
Designing, Building, and Deploying Messaging Solutions, Gregor Hohpe');

INSERT INTO `livraria`.`users` (`user_name`, `user_pass`) VALUES ('admin', '123');
INSERT INTO `livraria`.`users` (`user_name`, `user_pass`) VALUES ('usuario', '123');

INSERT INTO `livraria`.`user_roles` (`role_name`, `user_name`) VALUES
('administradores','admin');
INSERT INTO `livraria`.`user_roles` (`role_name`, `user_name`) VALUES
('usuarios','admin');
INSERT INTO `livraria`.`user_roles` (`role_name`, `user_name`) VALUES
('usuarios','usuario');
```

Listagem 3. Conteúdo do arquivo tomee.xml.

```
<tomee>
  <Resource id="dsLivraria" type="DataSource">
    JdbcDriver com.mysql.jdbc.Driver
    JdbcUrl jdbc:mysql://localhost:3306/livraria
    UserName root
    Password master
  </Resource>
</tomee>
```

Para que o Eclipse compile o módulo EJB e faça sua distribuição junto com as aplicações web, é preciso vinculá-lo aos demais projetos. A tela que aparece na **Figura 4**, acessada através das propriedades do projeto LivrariaWEB, mostra como estabelecer a conexão. Para realizá-la, estando com o ambiente de desenvolvimento aberto e o projeto selecionado, clique com o botão direito do mouse sobre ele e acesse *Properties* > *Deployment Assembly*. Essa mesma configuração também deve ser feita no projeto LivrariaWS. Ademais, é necessário adicionar o módulo EJB no *Java Build Path* do projeto LivrariaWEB, para que seja possível utilizar as classes durante a codificação. Isso pode ser feito clicando com o botão direito sobre esse projeto e acessando a opção *Build Path* > *Configure Build Path...*.

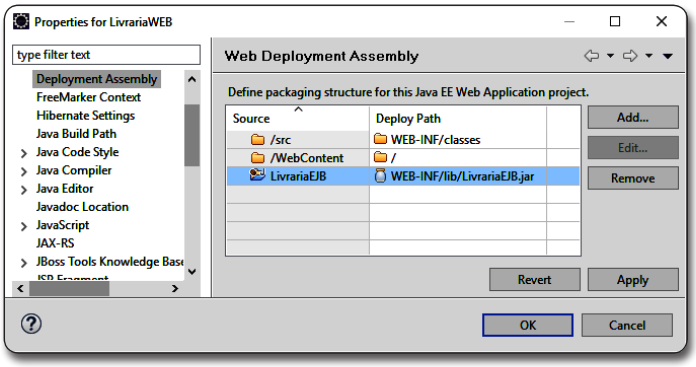

Figura 4. Vinculação do módulo EJB

Fluxo de funcionamento da aplicação

A página inicial da aplicação, exibida na **Figura 5**, apresenta uma lista pública de títulos e corresponde à opção *Home* do menu. As demais páginas, que podem ser acessadas através das opções *Cadastro JSF*, *Cadastro REST* e *Administração*, são de acesso restrito e só podem ser visualizadas por usuários autorizados. Assim, se após a abertura da aplicação o usuário tentar acessar qualquer uma dessas páginas, ele será redirecionado automaticamente para a tela de login, que pode ser vista na **Figura 6**.

Neste ponto é importante observar que existem três perfis de usuários na aplicação: anônimo, que só pode acessar a página principal; padrão, que permite acessar a tela *Cadastro JSF*, vista na **Figura 7**, bem como fazer a inclusão de títulos; e o perfil administrativo, que possibilita incluir, alterar e excluir títulos (vide **Figura 8**). Neste caso, se um usuário padrão digitar login e senha válidos, ele terá acesso às telas de cadastro. Porém, se após a autenticação com o perfil padrão ele tentar acessar a área administrativa, será apresentada a tela de acesso negado por falta de permissão, como sinaliza a **Figura 9**.

Estrutura dos projetos e arquivos que compõem a solução

Conforme mencionado anteriormente, LivrariaWS é uma instância de um projeto web que disponibiliza os web services da aplicação e possui somente o arquivo descritor de distribuição *web .xml*, cujo objetivo é definir o tipo de autenticação BASIC para os serviços e será discutido em detalhes no tópico sobre autenticação

e autorização com JAAS; por isso não foi apresentado na **Figura 10**. Nesta figura temos os módulos LivrariaEJB, que implementa as regras de negócio da solução, e LivrariaWEB, que utiliza *JavaServer Faces* (JSF) para criar as telas.

APIs utilizadas na aplicação

Agora que já vimos a configuração do ambiente, temos uma visão geral da arquitetura da aplicação e conhecemos seu fluxo de funcionamento, podemos analisar em detalhes as APIs utilizadas para desenvolvê-la.

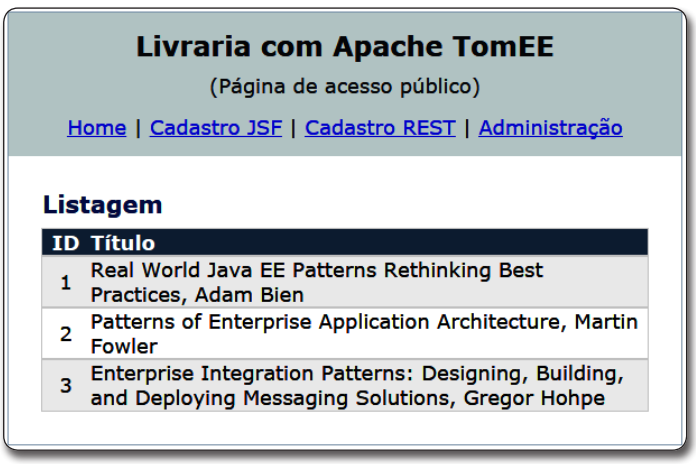

Figura 5. Tela principal da Livraria EE

Figura 6. Tela de login

Figura 7. Tela de cadastro

POJO com anotações JPA e Bean Validation

Vamos iniciar nossas considerações pelos componentes do projeto LivrariaEJB. Nele, temos a classe **Livro**, que utiliza anotações JPA, JAXB e validações conforme demonstra a **Listagem 4**. As anotações JPA são utilizadas para realizar o mapeamento da classe Java para a tabela *livro* do banco de dados. Por outro lado, a anotação **@XmlRootElement** é responsável por converter a lista-

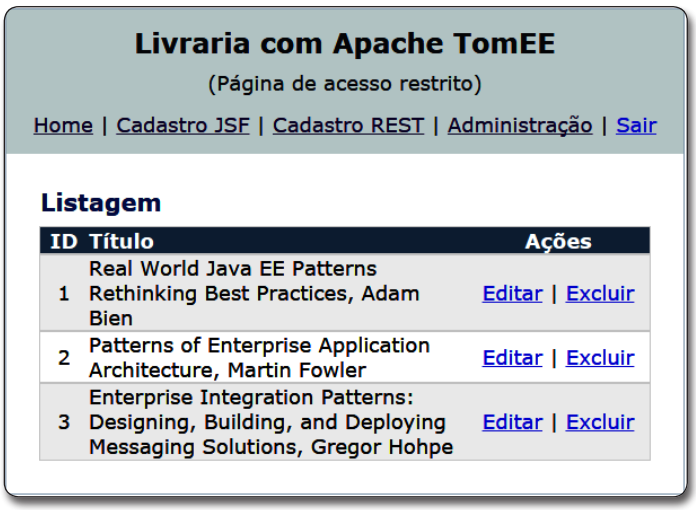

Figura 8. Tela administrativa

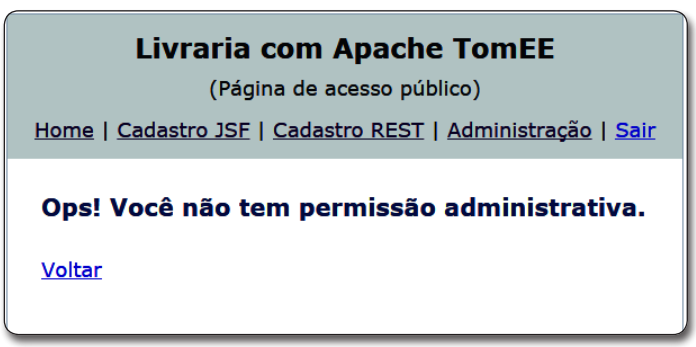

Figura 9. Tela de acesso negado

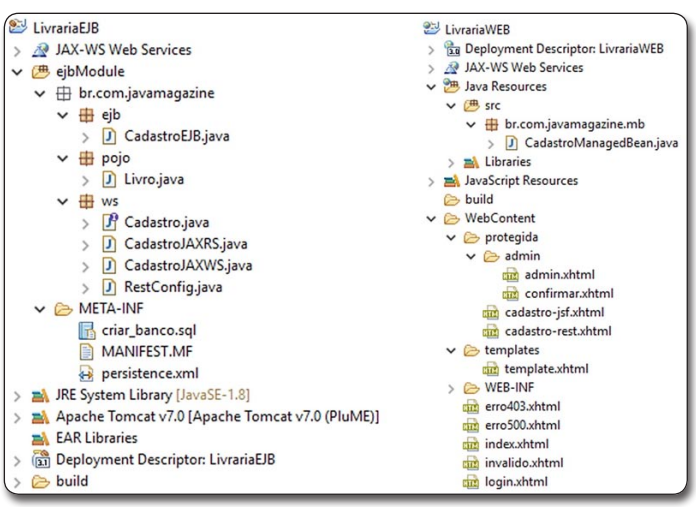

Figura 10. Estrutura dos projetos, pastas e arquivos

gem de livros para as representações XML ou JSON através da *Java Architecture for XML Binding* (JAXB) quando o usuário faz requisições ao web service. As validações, por sua vez, ocorrem em função da anotação **@NotEmpty**, quando o usuário está cadastrando ou alterando títulos. Caso durante o cadastro não for digitado nenhum valor, a mensagem da anotação é apresentada na tela para informar que o campo é obrigatório, conforme expõe a **Figura 7**.

Listagem 4. Código da classe Livro.

```
//Pacote e imports omitidos.

@Entity
@XmlRootElement
public class Livro implements Serializable {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private long id;

  @NotEmpty(message="Nome é obrigatório!")
  private String nome;

  //Getters e Setters.
}
```

Utilizando EJBs e gerenciamento de transações com JTA

Ainda no projeto LivrariaEJB, a classe **CadastroEJB**, apresentada na **Listagem 5**, disponibiliza o serviço de cadastro de títulos para os demais módulos através da anotação **@Stateless**, parte da especificação *Enterprise JavaBeans* que serve para criar um EJB local sem estado. No projeto, ela desempenha o papel de componente de negócios e ao mesmo tempo realiza as operações de persistência utilizando o **EntityManager**. Em um projeto real, é interessante criar uma camada intermediária para acesso a dados e isolamento das operações de persistência. A separação de interesses através de camadas, neste caso entre as operações de dados e as regras de negócio, é considerada uma boa prática na arquitetura de aplicações por promover o desacoplamento, facilitar a reusabilidade e manutenibilidade do código.

Conforme citado no parágrafo anterior, as operações de persistência são realizadas pelo **EntityManager** que é injetado no componente EJB com a anotação **@PersistenceContext** – parte da especificação JPA. Para que sejam realizadas as operações, é criada a unidade de persistência de dados **cadastro** no arquivo *persistence.xml*, apresentada na **Listagem 6** e referenciada no corpo da anotação **@PersistenceContext** na listagem anterior à mesma. Esse arquivo deve ficar dentro da pasta *META-INF* do módulo EJB e é responsável por conectar o contexto de persistência ao gerenciador de transações do TomEE. Finalizando as configurações, a tag **<jta-data-source>** faz referência à fonte de dados **dsLivraria**, definida no arquivo *tomee.xml* e já exposta na **Listagem 3**.

Anotação para injeção de dependências e JavaServer Faces

Com o serviço de cadastro de títulos concluído, vamos consumi-lo na classe **CadastroManagedBean**, do módulo LivrariaWEB.

Listagem 5. Código da classe CadastroEJB.

```
//Pacote e imports omitidos.

@Stateless
public class CadastroEJB {

  @PersistenceContext(unitName ="cadastro")
  private EntityManager em;

  public Livro buscarPorId(long id) {
    return em.find(Livro.class, id);
  }

  public void excluir(long id) {
    Livro livro = em.find(Livro.class, id);
    em.remove(livro);
  }

  public List<Livro> listar() {
    Query query = em.createQuery("select l from Livro as l");
    return query.getResultList();
  }

  public void salvar(String titulo) {
    salvar(new Livro(titulo));
  }

  public void salvar(Livro livro) {
    if (livro.getId() > 0) {
      em.merge(livro);
    } else {
      em.persist(livro);
    }
  }
}
```

Listagem 6. Código do arquivo persistence.xml.

```
<persistence ...>
  <persistence-unit name="cadastro">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>dsLivraria</jta-data-source>
  </persistence-unit>
</persistence>
```

Para isso, uma instância do componente **CadastroEJB** está sendo injetada no *bean* gerenciado do JSF com a anotação **@EJB**. Até o Java EE 6 os *Enterprise JavaBeans* eram objetos complexos, difíceis de implementar e manipular, mas a partir desta versão sua utilização se tornou mais fácil, uma vez que, além de outras mudanças, os EJBs passaram a ser classes Java básicas, chamadas de *Plain Old Java Objects*. Encerrando a análise sobre as configurações feitas no módulo web, temos as anotações **@ManagedBean** e **@SessionScoped**, observadas na **Listagem 7**, que servem, respectivamente, para definir um componente gerenciado pelo contêiner JSF e para especificar seu escopo.

Serviços web com anotações JAX-WS e JAX-RS

A utilização de web services tornou-se uma forma muito comum de implementar arquiteturas orientadas a serviço (SOA) e de realizar a integração de sistemas. Sua aceitação deve-se, principalmente, ao ganho de interoperabilidade entre as diferentes linguagens e plataformas existentes.

Nota

Como não faz parte do escopo desse artigo, não iremos abordar a construção das páginas JSF. Contudo, os interessados poderão analisá-las no código fonte disponibilizado junto com a revista.

Listagem 7. Código da classe CadastroManagedBean.

```
//Pacote e imports omitidos.

@SessionScoped
@ManagedBean(name = "cadastro")
public class CadastroManagedBean {

   @EJB
   private CadastroEJB cadastro;

   //Métodos para lidar com as páginas JSF.
}
```

Com a evolução da computação distribuída em dispositivos móveis e o aumento expressivo do número de usuários nesta plataforma, mais do que nunca projetar aplicações que exponham os serviços através da web é um requisito fundamental para soluções modernas.

Diante dessa necessidade, a plataforma Java EE define duas APIs de integração através de serviços web: *Java API for XML Web Services* (JAX-WS) e *Java API for RESTful Web Services* (JAX-RS). A primeira é baseada no protocolo SOAP, que define um modelo mais rígido de troca de mensagens, sendo mais utilizado em projetos de médio e grande porte para expor seus serviços através de arquiteturas orientada a serviços. Já a segunda, baseada na tese de doutorado de Roy Fielding sobre estilos arquiteturais e softwares distribuídos, publicada em 2000, define uma arquitetura de serviços mais flexível e permite a troca de mensagens em diferentes formatos. Conhecido apenas como REST, o modelo de disponibilização de serviços proposto por Fielding tem sido amplamente utilizado na integração de aplicações corporativas com aplicativos de dispositivos móveis por ser menos verboso e possibilitar um modelo mais simplificado e leve de troca de mensagens.

Através do TomEE é possível expor os serviços utilizando qualquer uma das abordagens. Assim, visando explorar essas possibilidades, os serviços de nosso exemplo serão expostos através do projeto LivrariaWS em ambos os formatos, SOAP e REST.

Iniciando a análise com a classe **CadastroJAXWS**, apresentada na **Listagem 8**, o serviço no formato SOAP é disponibilizado com o uso da anotação **@WebService** – parte da especificação JAX-WS – e através da implementação da interface **Cadastro**. Com relação ao serviço REST, utilizamos na classe **CadastroJAXRS** as anotações **@Path**, **@Produces**, **@GET**, **@POST** e **@DELETE** – como expõe a **Listagem 9** – que fazem parte da especificação JAX-RS e servem para indicar quais métodos da classe serão expostos, qual tipo de requisição eles irão atender e o caminho em que estarão disponíveis. As demais anotações nessas duas classes estão relacionadas aos aspectos de segurança da aplicação e serão discutidas no próximo tópico.

Listagem 8. Código da classe CadastroJAXWS.

```
//Pacote e imports omitidos.

@Stateless(name="cadastro")
@DeclareRoles({"usuarios","administradores"})
@WebService(portName = "cadastroPort",
   serviceName = "cadastroService",
   targetNamespace = "http://devmedia.com.br/wsdl",
   endpointInterface = "br.com.javamagazine.ws.Cadastro")
public class CadastroJAXWS implements Cadastro {

   @EJB
   private CadastroEJB cadastro;

   public Livro buscarPorId(long id) {
      return cadastro.buscarPorId(id);
   }

   public List<Livro> listar() {
      return cadastro.listar();
   }
   @RolesAllowed({"usuarios","administradores"})
   public void salvar(long id, String titulo) {
      cadastro.salvar(new Livro(id, titulo));
   }
   @RolesAllowed("administradores")
   public void excluir(long id) {
      cadastro.excluir(id);
   }
}
```

Listagem 9. Código da classe CadastroJAXRS.

```
//Pacote e imports omitidos.

@Stateless(name="cadastro")
@Path("/cadastro")
@DeclareRoles({"usuarios","administradores"})
@Produces({"application/json; charset=UTF-8"})
public class CadastroJAXRS {

   @EJB
   private CadastroEJB cadastro;

   @GET
   @Path("/buscar/{id}")
   public Livro buscarPorId(@PathParam("id") long id) {
      return cadastro.buscarPorId(id);
   }

   @GET
   @Path("/listar")
   public List<Livro> listar() {
      return cadastro.listar();
   }

   @POST
   @Path("/salvar/{id}")
   @RolesAllowed({"usuarios","administradores"})
   public Response salvar(@PathParam("id") long id, @FormParam("titulo")
   String titulo) {
      Livro livro = new Livro(id, titulo);
      cadastro.salvar(livro);
      return Response.status(200).entity(livro).build();
   }

   @DELETE
   @Path("/excluir/{id}")
   @RolesAllowed("administradores")
   public Response excluir(@PathParam("id") long id) {
      cadastro.excluir(id);
      return Response.status(200).build();
   }
}
```

Embora não faça parte do escopo desse artigo analisar em detalhes a implementação de web services, vamos considerar mais alguns aspectos de sua publicação no Apache TomEE. Quando disponibilizamos os serviços, normalmente criamos URLs referentes aos endereços de rede onde eles serão publicados.

No caso do serviço SOAP, a URL criada foi *http://localhost:8080/webservices/soap/cadastro?wsdl* e para o serviço REST, *http://localhost:8080/webservices/rest/cadastro*. Para isso, foram feitas as seguintes configurações:

1. O contexto *webservices* foi configurado no projeto LivrariaWS em *Properties* > *Web Project Settings* > *Context Root*. Ao fazer a distribuição da aplicação, independentemente da IDE, o nome do contexto também pode ser determinado no servidor através de um arquivo de configuração ou apenas renomeando o arquivo WAR para o nome de contexto desejado;
2. Após a configuração do contexto, é preciso definir o caminho em que o serviço será acessado. No caso do serviço SOAP, a localização especificada foi *soap* e está registrada no arquivo *system.properties* (vide **Listagem 10**). Esse arquivo fica na pasta *conf* do Apache TomEE e possui mais algumas configurações que informam ao TomEE o nome do serviço;
3. Por fim, para configurar o caminho do serviço REST, criamos uma classe que estende a classe **Application**. No exemplo, a classe **RestConfig**, observada na **Listagem 11**. Nela, utilizamos a anotação **@ApplicationPath** para definir a localização *rest* como caminho principal do serviço.

É importante observar que, de acordo com as configurações realizadas, no caso dos serviços JAX-WS o nome do serviço é o mesmo nome dado ao componente EJB através da anotação **@Stateless**. Já no caso do JAX-RS o nome é definido com a anotação **@Path**.

Nota

Para testar os serviços SOAP e RESTful, utilize a ferramenta SOAP-UI. No site da ferramenta há páginas que mostram como ajustar a ferramenta para acessar os serviços com os tipos de autenticação BASIC e FORM.

Listagem 10. Conteúdo do arquivo system.properties.

```
tomee.jaxws.subcontext = /soap
openejb.deploymentId.format = {appId}/{ejbName}
openejb.wsAddress.format = /{ejbName}
```

Listagem 11. Código da classe RestConfig.

```
//Pacote e imports omitidos.

@ApplicationPath("/rest")
public class RestConfig extends Application {
   public Set<Class<?>> getClasses() {
      return new HashSet<Class<?>>(Arrays.asList(CadastroJAXRS.class));
   }
}
```

Autenticação e autorização com JAAS

Um dos principais requisitos não-funcionais no desenvolvimento de uma aplicação corporativa é a segurança dos dados. Garantir que o usuário que está acessando o sistema é quem diz ser e assegurar que ele acesse apenas seus dados envolve duas etapas: autenticação e autorização.

A autenticação é a etapa que verifica se o usuário é realmente quem diz ser. Ela ocorre antes da autorização e é realizada normalmente pela confirmação de um usuário e senha. A autorização, por sua vez, assegura que o usuário está acessando somente os recursos e funcionalidades definidas para o seu perfil.

A fim de prover autenticação e autorização, o Tomcat inclui recursos desenvolvidos com base na API *Java Authentication and Authorization Service* (JAAS), parte integrante do Java SE desde a versão 1.4 do JDK. Assim, ao desenvolver o TomEE, nenhum módulo adicional de segurança foi acrescentado; ao contrário, todos os recursos disponíveis no Tomcat foram aproveitados integralmente. Seguindo a estrutura dele, as informações de usuário, senha e os perfis são obtidos nos *realms*, que nada mais são do que representações da localização onde as credenciais estão armazenadas.

Os realms são disponibilizados pelo Tomcat através da interface **org.apache.catalina.Realm** e algumas implementações para simplificar o acesso aos dados. Isto ocorre porque, embora a documentação da Servlet apresente uma forma bastante flexível para especificar os requisitos de segurança de uma aplicação através do descritor *web.xml*, não existe nenhuma API ou padrão que diga como integrar o servidor ao local de armazenamento das credenciais, sendo assim responsabilidade de cada servidor definir como será o processo de autenticação e autorização. Deste modo, as implementações disponíveis, são:

- **JDBCRealm:** realiza a autenticação em banco de dados relacional através do driver JDBC;
- **DataSourceRealm:** acessa as informações através de uma fonte JNDI JDBC;
- **UserDatabaseRealm:** acessa as informações em documentos XML (ex.: *tomcat-users.xml*);
- **JNDIRealm:** acessa os dados em servidores LDAP;
- **MemoryRealm:** armazena e acessa os dados em memória;
- **JAASRealm:** acessa as informações através do framework de autenticação e autorização (JAAS);
- **CombinedRealm:** acessa e autentica o usuário através de um ou mais Realms;
- **LockOutRealm:** é uma implementação que estende **CombinedRealm** e serve para bloquear tentativas de acesso através de ataques de força bruta.

Visto que o Apache TomEE não possui um recurso específico para manter o cadastro de usuários, senhas e perfis, cabe ao programador escolher um dos *realms* ou desenvolver uma implementação própria e criar o repositório das credenciais dos usuários. Neste exemplo foi utilizado o **DataSourceRealm**, disponível com o Tomcat, para apontar a fonte de dados do MySQL onde elas foram

armazenadas (veja **Listagem 12**). Lembre-se que as informações de autenticação, incluindo usuários, senhas e os perfis de cada um, foram inseridas no banco de dados com o script da **Listagem 2**.

Voltando à **Figura 3**, note que na arquitetura da Livraria EE existem alguns casos de uso que requerem autenticação e autorização (veja os fluxos 1, 2, 3 e 4). Dentre eles, verificamos que no fluxo 4 ocorre a conexão entre o módulo LivrariaEJB e a camada de dados. Como a autenticação nesse fluxo é gerenciada pelo SGBD, que possui um mecanismo de autenticação e autorização próprio, não vamos nos preocupar com ele. Nos demais fluxos, no entanto, a proteção dos serviços e informações é responsabilidade da aplicação e, portanto, vamos analisá-los minuciosamente.

Nota

Um cuidado especial para quem vai rodar o projeto no Eclipse é que as configurações apresentadas na **Listagem 12** devem ser feitas no arquivo server.xml do ambiente de desenvolvimento. Esse arquivo fica dentro da pasta Servers, gerada automaticamente na aba Project Explorer quando o servidor é adicionado.

Listagem 12. Conteúdo do arquivo server.xml.

```
<Realm className="org.apache.catalina.realm.DataSourceRealm"
  dataSourceName="dsLivraria"
  roleNameCol="role_name"
  userCredCol="user_pass"
  userNameCol="user_name"
  userRoleTable="user_roles"
  userTable="users" />
```

Ao acessar os recursos protegidos da aplicação – fluxos 1, 2 e 3 – o usuário precisa estar autenticado. Assim, quando ele solicita o recurso, o TomEE intercepta a requisição e verifica se ele está logado e possui permissões suficientes para acessá-lo, com base nas configurações feitas nos arquivos *web.xml* dos projetos LivrariaWEB e LivrariaWS (ver **Listagens 13** e **14**). Quando o usuário não está autenticado ou não possui as permissões necessárias, ele é redirecionado automaticamente para a tela de login ou de acesso negado.

Ainda considerando os aspectos de segurança da Livraria EE, ao fazer uma análise mais criteriosa da arquitetura apresentada na **Figura 3** o leitor poderia se perguntar: Não seria possível publicar os serviços do módulo LivrariaWS através da aplicação web, evitando assim a criação de dois módulos? A resposta para essa pergunta é sim, os serviços poderiam ser disponibilizados neste módulo. Porém, o projeto web utiliza autenticação do tipo FORM e também precisávamos disponibilizar os serviços no formato BASIC. Como não é permitido mais do que um tipo de autenticação por contexto, para resolver esse problema criamos a aplicação LivrariaWS, com um novo contexto utilizando o tipo BASIC. Assim, temos duas versões de web services: uma que aceita o tipo de autenticação BASIC e outra que aceita o tipo FORM.

No módulo LivrariaWEB, é possível cadastrar títulos através do web service REST publicado em seu contexto utilizando o método POST e a autenticação FORM. Com isso, é criada uma sessão durável, gerenciada pelo TomEE e que só expira por *timeout* ou quando for invalidada. Dessa forma o usuário pode submeter os dados aos web services desse módulo sem enviar as credenciais em cada requisição.

Como podemos observar, ambos os tipos podem ser usados independentemente no TomEE, seja em aplicações web com páginas processadas do lado servidor, com JSP/JSF, seja em aplicações web com páginas processadas do lado cliente, com JavaScript. O tipo FORM tem a vantagem de permitir a definição do layout da tela de login, mas não possibilita a autenticação em aplicações que rodam fora dos navegadores, a não ser que seja criado um mecanismo para submeter as credenciais primeiro e depois manter a sessão HTTP ativa. O tipo BASIC, por sua vez, tem a desvantagem de não permitir a customização da tela de login nos navegadores, mas não requer autenticação prévia. Assim, as requisições de clientes que não sejam navegadores podem ser feitas com o envio das credenciais do usuário em cada requisição.

Com relação à autorização, os recursos são protegidos através de configurações feitas no arquivo *web.xml* (observe novamente as **Listagens 13** e **14**). Embora existam várias tags nesse arquivo, as três principais são: **<url-pattern>**, que define as URLs a serem interceptadas pelo TomEE e que precisam de autorização; **<http-method>**, que reforça a segurança identificando quais métodos invocados precisam de autorização; e **<auth-constraint>**, que vincula os perfis que terão acesso aos recursos requisitados nas URLs.

Para concluir os ajustes necessários para viabilizar a autorização, além de definir URLs, métodos HTTP e perfis no arquivo *web.xml*, é preciso configurar as classes **CadastroJAXWS** e **CadastroJAXRS** (vide **Listagens 8** e **9**) com **@DeclareRoles** para informar quais perfis serão gerenciados pelo Apache TomEE. Ademais, deve-se utilizar a anotação **@RolesAllowed** para determinar quais perfis podem acessar os serviços ou **@PermitAll** para eliminar as restrições e **@DenyAll** para restringir totalmente o acesso. Essas anotações são especificadas na *Common Annotations for the Java Platform* e servem para proteger os métodos das classes citadas, transferindo a responsabilidade de interceptar as requisições, autenticar e autorizar o acesso para o TomEE.

Finalizada a codificação, o próximo passo é compilar os projetos e então publicá-los. Como poderá ser observado, a distribuição de aplicações no TomEE é tão simples quanto no Tomcat: basta gerar os arquivos .war e colocá-los na pasta *webapps*. Um ponto importante com relação aos arquivos de distribuição da solução Livraria EE diz respeito ao tamanho. Note que o arquivo *LivrariaWEB.war* ficou com apenas 16.2 KB, e o *LivrariaWS.war*, com 7.13 KB. Vale salientar que em ambos já está incluído o serviço EJB do módulo LivrariaEJB. Obviamente, embora o tamanho dos arquivos seja parcialmente justificado pela simplicidade da solução, eles ficaram pequenos assim principalmente pelo fato de não precisarem acrescentar bibliotecas adicionais na distribuição, já que as implementações das APIs fazem parte do servidor Apache TomEE.

Listagem 13. Fragmento do web.xml que cuida dos aspectos de segurança da LivrariaWeb.

```
<web-app ...>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Somente administradores</web-resource-name>
      <url-pattern>/protegida/admin/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>administradores</role-name>
    </auth-constraint>
  </security-constraint>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Usuários e administradores</web-resource-name>
      <url-pattern>/protegida/*</url-pattern>
      <url-pattern>/soap/*</url-pattern>
      <url-pattern>/rest/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>administradores</role-name>
      <role-name>usuarios</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <role-name>administradores</role-name>
  </security-role>
  <security-role>
    <role-name>usuarios</role-name>
  </security-role>
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.xhtml</form-login-page>
      <form-error-page>/invalido.xhtml</form-error-page>
    </form-login-config>
  </login-config>
</web-app>
```

Listagem 14. Fragmento do web.xml que cuida dos aspectos de segurança da LivrariaWS.

```
<web-app ...>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Usuários e administradores</web-resource-name>
      <url-pattern>/soap/*</url-pattern>
      <url-pattern>/rest/*</url-pattern>
    <http-method>POST</http-method>
      <http-method>DELETE</http-method>
      <http-method>PUT</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>administradores</role-name>
      <role-name>usuarios</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <role-name>administradores</role-name>
  </security-role>
  <security-role>
    <role-name>usuarios</role-name>
  </security-role>
  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>
</web-app>
```

Para aqueles que já desenvolveram aplicações web, considerando todas as APIs utilizadas na aplicação de exemplo, quantas bibliotecas e dependências precisariam ser incluídas em um projeto web para que ele rodasse no Apache Tomcat e tivesse as mesmas funcionalidades implementadas na Livraria EE? Quais bibliotecas precisariam ser incluídas e qual seria o tamanho final das aplicações, considerando que elas seriam distribuídas juntamente com a solução?

Conforme citado, o exemplo desenvolvido neste artigo é modesto do ponto de vista funcional, mas mesmo assim requer do arquiteto ou desenvolvedor um conhecimento considerável sobre as tecnologias empregadas. No entanto, note que ao desenvolver uma solução para rodar num servidor de aplicações corporativas que segue as especificações da plataforma Java EE, neste caso o TomEE, nos preocupamos exclusivamente em utilizar os recursos da plataforma. Caso optássemos por desenvolver a solução adotando um servidor que implementasse apenas Servlet e JSP, teríamos que nos preocupar também em identificar quais bibliotecas fariam parte da solução, assegurar que elas fossem compatíveis e que funcionariam corretamente neste servidor. Agora, isso não é mais necessário àqueles que utilizam o Tomcat, pois o que a comunidade que desenvolve o TomEE tem feito é cuidar da integração das bibliotecas que implementam as APIs do Java EE e disponibilizá-las num servidor leve e robusto: o Apache TomEE.

Autor

Geucimar Brilhador
geucimar@gmail.com

Possui as certificações OCP-JP e OCP-WCD. É Professor do Bacharelado em Sistemas de Informação da Universidade Positivo, em Curitiba-PR, Especialista em Engenharia de Software pela Universidade Federal do Paraná (UFPR), Especialista em Java e Desenvolvimento para Dispositivos Móveis pela Universidade Tecnológica Federal do Paraná (UTFPR), formado em Sistemas de Informação pela UFPR.

Links:

Página do servidor Apache TomEE.
http://tomee.apache.org

Endereço para download do Java Development Kit.
http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html

Endereço para download do Eclipse.
https://eclipse.org/downloads

Tudo sobre Realm no Tomcat.
https://tomcat.apache.org/tomcat-7.0-doc/realm-howto.html

Segurança em web services REST.
https://jersey.java.net/documentation/latest/security.html

SoapUI para teste de web services.
https://www.soapui.org

Página de acesso à edição 146 da Java Magazine.
http://www.devmedia.com.br/revista-java-magazine-146/33970

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486