Tutorial

Construções para aplicativos de mundo real

Recursos de linguagem Java mais avançados

Por: J Steven Perry
Publicado 22/02/2016

Antes de iniciar

Descubra o que espera deste tutorial e como aproveitá-lo ao máximo.

Sobre este tutorial

O tutorial em duas partes *Introdução à programação de Java* é destinado a desenvolvedores de software que estão conhecendo a tecnologia Java. Conclua as duas partes para começar a utilizar a programação orientada a objetos (OOP) e o desenvolvimento de aplicativo para o mundo real usando a linguagem e a plataforma Java.

Esta segunda metade do tutorial *Introdução à programação de Java* introduz capacidades da linguagem Java que são mais sofisticadas do que as abordadas na Parte 1.

Objetivos

A linguagem Java é madura e sofisticada o suficiente para ajudá-lo a realizar praticamente qualquer tarefa de programação. Este tutorial apresenta os recursos da linguagem Java que são necessários para manipular cenários de programação complexos, incluindo:

- Manipulação de exceção
- Herança e abstração
- Interfaces
- Classes aninhadas
- Expressões regulares
- Tipos
- `enum` genéricos

- E/S
- Serialização

Pré-requisitos

O conteúdo deste tutorial é direcionado para programadores que estão conhecendo a linguagem Java e não estão familiarizados com seus recursos mais sofisticados. O tutorial pressupõe que você concluiu "*Introdução à programação de Java*, Parte 1: Princípios básicos da linguagem Java" para:

- Obter um entendimento dos princípios básicos da OOP na plataforma Java
- Configurar o ambiente de desenvolvimento para os exemplos do tutorial
- Iniciar o projeto de programação que continuará desenvolvendo na Parte 2

Requisitos do sistema

Para concluir os exercícios deste tutorial, instale e configure um ambiente de desenvolvimento composto por:

- JDK 8 da Oracle
- IDE do Eclipse para desenvolvedores de Java

As instruções de download e instalação de ambos estão incluídas na Parte 1.

A configuração do sistema recomendada é:

- Um sistema com suporte para Java SE 8 com pelo menos 2 GB de memória. Há suporte para Java 8 Linux®, Windows®, Solaris® e Mac OS X.
- Pelo menos 200 MB de espaço em disco para instalar os componentes de software e exemplos.

Próximas etapas com objetos

Parte 1 deste tutorial terminaram com uma classe `Person` que era razoavelmente útil, mas poderia ser mais. Aqui, você começa a aprender sobre técnicas para aprimorar uma classe como `Person`, iniciando com as técnicas a seguir:

- Sobrecarregando métodos
- Substituindo métodos
- Comparando um objeto com outro
- Deixando seu código mais fácil de depurar

Sobrecarregando métodos

Quando dois métodos são criados com o mesmo nome, mas com listas de argumentos diferentes (ou seja, números ou tipos de parâmetros diferentes), você tem um *sobrecarregado*. No tempo de

execução, o Java Runtime Environment (JRE; também conhecido como Java Runtime) decide qual variação do seu método sobrecarregado deve chamar com base nos argumentos que foram passados para ele.

Suponha que `Person` precisa de alguns métodos para imprimir uma auditoria do seu estado atual. Chamo esses métodos de `printAudit()`. Cole o método sobrecarregado na Listagem 1 na visualização do editor do Eclipse em `Person`:

`printAudit()`: Um método sobrecarregado

```
public void printAudit(StringBuilder buffer) {
  buffer.append("Name="); buffer.append(getName());
  buffer.append(","); buffer.append("Age="); buffer.append(getAge());
  buffer.append(","); buffer.append("Height="); buffer.append(getHeight());
  buffer.append(","); buffer.append("Weight="); buffer.append(getWeight());
  buffer.append(","); buffer.append("EyeColor="); buffer.append(getEyeColor());
  buffer.append(","); buffer.append("Gender="); buffer.append(getGender());
}
public void printAudit(Logger l) {
  StringBuilder sb = new StringBuilder();
  printAudit(sb);
  l.info(sb.toString());
}
```

Mostrar mais

Você tem duas versões sobrecarregadas de `printAudit()` e uma inclusive usa a outra. Ao fornecer duas versões, você permite que o responsável pela chamada escolha como imprimir uma auditoria da classe. Dependendo dos parâmetros que são passados, o Java Runtime exige o método correto.

Duas regras de sobrecarga de método

Lembre-se destas duas regras importantes ao usar métodos sobrecarregados:

- Não é possível sobrecarregar um método apenas mudando seu tipo de retorno.
- Não é possível ter dois métodos com o mesmo nome com a mesma lista de parâmetros.

Se você violar essas regras, o compilador fornece um erro.

Substituindo métodos

Quando uma subclasse de outra classe fornece sua própria implementação de um método definido em uma classe-pai, isso é chamado de *substituição de método*. Para ver como a substituição de método é útil, você precisa trabalhar um pouco no seu `Employee`. Depois de configurar, mostrarei como a substituição de método pode ser útil.

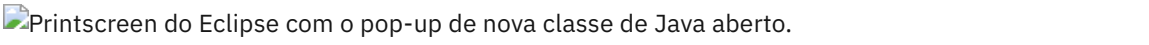
`Employee`: Uma subclasse de `Person`

Lembre-se que na Parte 1 deste tutorial que `Employee` pode ser uma subclasse (ou *filho*) de `Person` que possui alguns atributos adicionais:

- Número de identificação do contribuinte
- Número de matrícula
- Data de admissão
- Salário

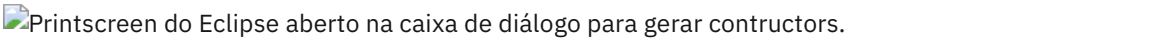
Para declarar tal classe em um arquivo chamado Employee.java, clique com o botão direito no pacote `com.makotojava.intro` no Eclipse. Clique em **New > Class...** para abrir a caixa de diálogo New Java Class, mostrada na Figura 1.

Caixa de diálogo New Java Class


Printscreen do Eclipse com o pop-up de nova classe de Java aberto.

Insira `Employee` como nome da classe e `Person` como sua superclasse; depois clique em **Concluir**. É possível ver a classe `Employee` em uma janela de edição. Não é preciso declarar explicitamente um construtor, mas vá em frente e implemente os dois construtores de qualquer forma. Em primeiro lugar, certifique-se de que a janela de edição de classe `Employee` tenha o foco e acesse **Source > Generate Constructors from Superclass...**. Você verá uma caixa de diálogo em que é possível selecionar os construtores para implementar, tal como mostrado na Figura 2.

Caixa de diálogo Generate Constructors from Superclass


Printscreen do Eclipse aberto na caixa de diálogo para gerar contructors.

Selecione os dois construtores (como mostrado na Figura 2) e clique em **OK**. O Eclipse gera os construtores para você. Agora você tem uma classe `Employee` como a da Listagem 2

A nova classe `Employee` aprimorada

```
package com.makotojava.intro;

public class Employee extends Person {

  public Employee() {
    super();
    // TODO Auto-generated constructor stub
  }

  public Employee(String name, int age, int height, int weight,
  String eyeColor, String gender) {
    super(name, age, height, weight, eyeColor, gender);
    // TODO Auto-generated constructor stub
  }
```

Mostrar mais

`Employee` herda de `Person`

`Employee` herda os atributos e o comportamento do seu pai, `Person`, e também tem algo próprio, como pode ser visto na Listagem 3.

A classe `Employee` com os atributos de `Person`.

```
package com.makotojava.intro;

import java.math.BigDecimal;

public class Employee extends Person {

  private String taxpayerIdentificationNumber;
  private String employeeNumber;
  private BigDecimal salary;

  public Employee() {
    super();
  }
  public String getTaxpayerIdentificationNumber() {
    return taxpayerIdentificationNumber;
```

Mostrar mais

Não se esqueça de gerar getters e configuradores para os novos atributos. Você viu como fazer isso na Parte 1.

Substituição de método: `printAudit()`

Agora, como prometido, você está pronto para um exercício de métodos de substituição. Você substituirá o método `printAudit()` (consulte a `printAudit()`: Um método sobrecarregado) usada para formatar o estado atual de uma instância `Person`. `Employee` herda esse comportamento de `Person`. Se você instanciar `Employee`, configurar seus atributos e chamar uma das sobrecargas de `printAudit()`, a chamada será bem-sucedida. Entretanto, a auditoria que é produzida não representará totalmente um `Employee`. O problema é que `printAudit()` não pode formatar os atributos específicos de um `Employee`, porque `Person` não sabe sobre eles.

A solução é substituir a sobrecarga de `printAudit()` que pega um `StringBuilder` como parâmetro e inclui um código para imprimir os atributos específicos de `Employee`.

Para implementar essa solução no seu IDE do Eclipse, verifique se `Employee` está aberto na janela do editor ou se foi selecionado na visualização Project Explorer. Em seguida, acesse **Source > Override/Implement Methods...**, e você verá uma caixa de diálogo, mostrada na Figura 3, na qual pode selecionar os métodos para substituir ou implementar.

Caixa de diálogo Override/Implement Methods


Printscreen do Eclipse aberto na caixa de diálogo para implementar ou substituir métodos.

Selecione o plug-in `StringBuilder` como sobrecarga de `printAudit()`, tal como mostrado na Figura 3, e clique em **OK**. O Eclipse gera o stub de método para você e, em seguida, é possível preencher o resto, deste modo:

```
@Override
public void printAudit(StringBuilder buffer) {
  // Call the superclass version of this method first to get its attribute values
  super.printAudit(buffer);
  // Now format this instance's values
  buffer.append("TaxpayerIdentificationNumber=");
  buffer.append(getTaxpayerIdentificationNumber());
  buffer.append(","); buffer.append("EmployeeNumber=");
  buffer.append(getEmployeeNumber());
```

```
    buffer.append(","); buffer.append("Salary=");
    buffer.append(getSalary().setScale(2).toPlainString());
}
```

Mostrar mais

Observe a chamada para `super.printAudit()`. Aqui, você está pedindo para a superclasse (`Person`) exibir seu comportamento para `printAudit()`; depois, você aumentará com o comportamento `Employee`-type `printAudit()`.

A chamada para `super.printAudit()` não precisa acontecer primeiro; apenas pareceu uma boa ideia imprimir esses atributos primeiro. Na verdade, você nem precisa chamar o `super.printAudit()`. Se não chamá-lo, deve-se formatar os atributos de `Person` por conta própria no `Employee.printAudit()`.

Membros da classe

As variáveis e métodos que você tem em `Person` e `Employee` são variáveis e métodos de *instância*. Para usá-los, você deve instanciar a classe de que precisa ou ter uma referência para a instância. Cada instância do objeto possui variáveis e métodos e, para cada uma delas, o comportamento exato será diferente, porque se baseia no estado da instância do objeto.

As próprias classes também podem ter variáveis e métodos. Você declara as variáveis de classes com a palavra-chave `static` introduzida na Parte 1. As diferenças entre variáveis de classes e variáveis de instância são:

- Cada instância de uma classe compartilha uma única cópia de uma variável de classes.
- É possível chamar métodos de classes na própria classe, sem ter uma instância.
- Os métodos de instância podem acessar variáveis de classes, mas os métodos de classes não podem acessar variáveis de instância.
- Os métodos de classes podem acessar somente variáveis de classes.

Incluindo variáveis e métodos de classes

Quando faz sentido incluir variáveis e métodos de classes? A melhor regra básica é fazer isso raramente, para não usar excessivamente. Dito isso, é uma boa ideia usar variáveis e métodos de classes:

- Para declarar constantes que qualquer instância da classe pode usar (e cujo valor é fixado no momento do desenvolvimento)
- Para controlar “contadores” de instâncias da classe
- Em uma classe com métodos de utilitário que não precisam sempre de uma instância da classe (como `Logger.getLogger()`)

Variáveis de classes

Para criar uma variável de classes, use a palavra-chave `static` quando declará-la:

```
accessSpecifier static variableName [= initialValue];
```

Mostrar mais

Observação: Aqui, os colchetes indicam que seus conteúdos são opcionais. Os colchetes não fazem parte da sintaxe da declaração.

O JRE cria um espaço na memória para armazenar cada variável de instância de classe para todas as instâncias dessa classe. Por outro lado, o JRE cria somente uma cópia de cada variável de classes, independentemente do número de instâncias. Ele faz isso na primeira vez que a classe é carregada (ou seja, na primeira vez que encontra a classe em um programa). Todas as instâncias da classe compartilham essa única cópia da variável. Desse modo, variáveis de classes são uma boa opção para constantes que todas as instâncias devem ser capazes de usar.

Por exemplo, você declarou o atributo `Gender` de `Person` como sendo do tipo `String`, mas não impôs restrições em torno dele. A Listagem 4 mostra um uso comum de variáveis de classes.

Usando variáveis de classes

```
public class Person {
  //. . .
  public static final String GENDER_MALE = "MALE";
  public static final String GENDER_FEMALE = "FEMALE";
  // . . .
  public static void main(String[] args) {
  Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", GENDER_MALE);
    // . . .
  }
  //. . .
}
```

Mostrar mais

Declarando constantes

Normalmente, as constantes são:

- Nomeadas somente em letras maiúsculas
- Nomeadas como diversas palavras, separadas por sublinhados
- Declaradas como `final` (para que seus valores não possam ser modificados)
- Declaradas com um especificador de acesso do tipo `public` (para que possam ser acessadas por outras classes que precisam fazer referência aos seus valores por nome)

Em Usando variáveis de classes, para usar a constante para `MALE` no `Person` como chamada do construtor, bastaria fazer uma referência ao seu nome. Para usar uma constante fora da classe, você criaria um prefácio com o nome da classe em que ela foi declarada:

```
String genderValue = Person.GENDER_MALE;
```

Mostrar mais

Métodos de classes

Se estiver acompanhando desde a Parte 1, você já chamou o método estático `Logger.getLogger()` várias vezes — sempre que recuperou uma instância `Logger` para escrever a saída no console. Observe, porém, que não precisou de uma instância de `Logger` para tal; em vez disso, fez uma referência à classe `Logger` propriamente dita. Esta é a sintaxe para fazer um *método de classes*. Tal como acontece com as variáveis de classes, a palavra-chave `static` identifica `Logger` (neste exemplo) como um método de classes. Às vezes, os métodos de classes também são chamados de *métodos estáticos* por esse motivo.

Usando métodos de classes

Agora, você combina o que aprendeu sobre variáveis e métodos estáticos para criar um método estático em `Employee`. Declara que uma variável `private static final` contém um `Logger`, que todas as instâncias compartilham e que pode ser acessado ao chamar `getLogger()` em `Employee`. A Listagem 5 mostra como.

Criando um método de classes (ou estático)

```
public class Employee extends Person {
  private static final Logger logger = Logger.getLogger(Employee.class.getName());
  //. . .
  public static Logger getLogger() {
    return logger;
  }

}
```

Mostrar mais

Duas coisas importantes estão acontecendo na Listagem 5:

- A instância `Logger` é declarada com acesso do tipo `private`; portanto, nenhuma classe fora de `Employee` pode acessar a referência diretamente.
- O `Logger` é inicializado quando a classe é carregada — porque a sintaxe do inicializador Java é usada para atribuir um valor.

Para recuperar a classe `Employee` com o objeto `Logger`, faça esta chamada:

```
Logger employeeLogger = Employee.getLogger();
```

Mostrar mais

Comparando objetos

A linguagem Java oferece duas maneiras de comparar objetos:

- O operador ==

- O método `equals()`

Comparando objetos com `==`

A sintaxe == compara objetos em termos de igualdade, de modo que `a == b` gera `true` somente quando a e b tiver o mesmo valor. No caso de objetos, ambos se referem à mesma instância do objeto. No caso de primitivas, os valores são idênticos. Suponha que um teste JUnit é gerado para `Employee` (o que você viu como fazer na Parte 1). O teste JUnit é mostrado na Listagem 6.

Comparando objetos com `==`

```
public class EmployeeTest {
  @Test
  public void test() {
    int int1 = 1;
    int int2 = 1;
    Logger l = Logger.getLogger(EmployeeTest.class.getName());

    l.info("Q: int1 == int2?          A: " + (int1 == int2));
    Integer integer1 = Integer.valueOf(int1);
    Integer integer2 = Integer.valueOf(int2);
    l.info("Q: Integer1 == Integer2?   A: " + (integer1 == integer2));
    integer1 = new Integer(int1);
    integer2 = new Integer(int2);
    l.info("Q: Integer1 == Integer2?   A: " + (integer1 == integer2));
    Employee employee1 = new Employee();
```

Mostrar mais

Se executar o código da Listagem 6 dentro do Eclipse (selecione `Employee` na visualização Project Explorer e, depois, selecione **Run As > JUnit Test**), a saída deve ser:

```
Sep 18, 2015 5:09:56 PM com.makotojava.intro.EmployeeTest test
INFO: Q: int1 == int2?          A: true
Sep 18, 2015 5:09:56 PM com.makotojava.intro.EmployeeTest test
INFO: Q: Integer1 == Integer2?   A: true
Sep 18, 2015 5:09:56 PM com.makotojava.intro.EmployeeTest test
INFO: Q: Integer1 == Integer2?   A: false
Sep 18, 2015 5:09:56 PM com.makotojava.intro.EmployeeTest test
INFO: Q: Employee1 == Employee2? A: false
```

Mostrar mais

No primeiro caso na Comparando objetos com ==, os valores das primitivas são os mesmos; logo, o operador == gera `true`. No segundo caso, os objetos `Integer` se referem à mesma instância e, novamente, == gera `true`. No terceiro caso, embora os objetos `Integer` agrupem o mesmo valor, == gera `false` porque `integer1` e `integer2` se referem a objetos diferentes. Pense em == como um teste para “mesmo objeto”.

Comparando objetos com `equals()`

`equals()` é um método que cada objeto de linguagem Java obtém gratuitamente, porque é definido como um método de instância de `java.lang.Object` (do qual cada objeto Java herda).

Chame `equals()` assim como chamaria qualquer outro método:

```
a.equals(b);
```

Mostrar mais

Essa declaração chama o método `equals()` do objeto `a`, passando para ele uma referência ao objeto `b`. Por padrão, um programa Java simplesmente verificaria se os dois objetos são iguais usando a sintaxe `==`. Mas, como `equals()` é um método, ele pode ser substituído. Compare o caso de teste JUnit na Comparando objetos com == com aquele da Listagem 7 (que chamei de `anotherTest()`), que utiliza `equals()` para comparar os dois objetos:

Comparando objetos com `equals()`

```
@Test
public void anotherTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
  Integer integer1 = Integer.valueOf(1);
  Integer integer2 = Integer.valueOf(1);
  l.info("Q: integer1 == integer2 ? A: " + (integer1 == integer2));
  l.info("Q: integer1.equals(integer2) ? A: " + integer1.equals(integer2));
  integer1 = new Integer(integer1);
  integer2 = new Integer(integer2);
  l.info("Q: integer1 == integer2 ? A: " + (integer1 == integer2));
  l.info("Q: integer1.equals(integer2) ? A: " + integer1.equals(integer2));
  Employee employee1 = new Employee();
  Employee employee2 = new Employee();
  l.info("Q: employee1 == employee2 ? A: " + (employee1 == employee2));
  l.info("Q: employee1.equals(employee2) ? A : " + employee1.equals(employee2));
```

Mostrar mais

A execução do código da Listagem 7 produz:

```
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: integer1 == integer2 ? A: true
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: integer1.equals(integer2) ? A: true
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: integer1 == integer2 ? A: false
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: integer1.equals(integer2) ? A: true
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: employee1 == employee2 ? A: false
Sep 19, 2015 10:11:57 AM com.makotojava.intro.EmployeeTest anotherTest
INFO: Q: employee1.equals(employee2) ? A : false
```

Mostrar mais

Uma observação sobre comparar `Integer`s

Na Comparando objetos com `equals()`, é de se esperar que o método `equals()` de `Integer` gere `true` se `==` gerar `true`. Observe, porém, o que acontece no segundo caso, em que é possível criar objetos separados que agrupam o valor 1: `==` gera `false` porque `integer1` e `integer2` se referem a objetos diferentes; mas `equals()` gera `true`.

Os gravadores do JDK decidiram que, para `Integer`, o significado de `equals()` seria diferente do padrão (que, como você deve lembrar, é comparar as referências do objeto para ver se elas se referem ao mesmo objeto). Para o `Integer`, `equals()` gera `true` em casos em que o valor `int` subjacente é o mesmo.

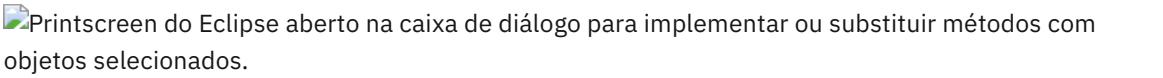
Para o `Employee`, você não substituiu `equals()`; portanto, o comportamento padrão (de usar ==) gera o esperado, porque `employee1` e `employee2` se referem a objetos diferentes.

Consequentemente, para qualquer objeto escrito, é possível definir o que `equals()` significa da forma adequada para o aplicativo que você está gravando.

Substituindo `equals()`

É possível definir o que `equals()` significa para os objetos do seu aplicativo substituindo o comportamento padrão de `Object.equals()`. Novamente, é possível usar o Eclipse para fazer isso. Certifique-se de que `Employee` tem o foco na sua janela de origem do IDE do Eclipse e, a seguir, acesse **Source > Override/Implement Methods**. A caixa de diálogo da Figura 4 é aberta.

Caixa de diálogo Override/Implement Methods


Printscreen do Eclipse aberto na caixa de diálogo para implementar ou substituir métodos com objetos selecionados.

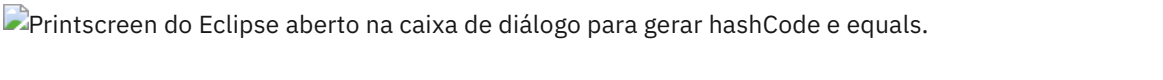
Essa caixa de diálogo foi utilizada antes, mas, neste caso, você deseja implementar o método de superclasse `Object.equals()`. Portanto, localize `Object` na lista de métodos para substituir ou implementar, selecione o método `equals(Object)` e clique em **OK**. O Eclipse gera o código correto e o coloca no seu arquivo de origem

Faz sentido que os dois objetos `Employee` sejam iguais se os estados de tais objetos forem iguais. Ou seja, são iguais se os valores — last name, first name, age — forem os mesmos.

Geração automática de `equals()`

O Eclipse pode gerar um método `equals()` para você com base nas variáveis de instância (atributos) definidas para uma classe. Como `Employee` é uma subclasse de `Person`, você primeiramente gera `equals()` para `Person`. Na visualização Project Explorer do Eclipse, clique com o botão direito em `Person` e escolha **Generate hashCode() e equals()** para abrir a caixa de diálogo mostrada na Figura 5, onde você seleciona os atributos que serão incluídos nos métodos `hashCode()` e `equals()`.

Caixa de diálogo Generate hashCode() e equals()


Printscreen do Eclipse aberto na caixa de diálogo para gerar hashCode e equals.

Selecione todos os atributos (tal como mostrado na Figura 5) e clique em **OK**. O Eclipse gera um método `equals()` que se parece com aquele da Listagem 8.

Um método `equals()` gerado pelo Eclipse

```
@Override
public boolean equals(Object obj) {
  if (this == obj)
    return true;
```

```
  if (obj == null)
    return false;
  if (getClass() != obj.getClass())
    return false;
  Person other = (Person) obj;
  if (age != other.age)
    return false;
  if (eyeColor == null) {
    if (other.eyeColor != null)
      return false;
  } else if (!eyeColor.equals(other.eyeColor))
```

Mostrar mais

Não se preocupe com `hashCode()` por enquanto — é possível mantê-lo ou exclui-lo. O método `equals()` gerado pelo Eclipse parece complicado, mas executa algo simples: se o objeto passado for o mesmo objeto que consta na Listagem 8, `equals()` gera `true`. Caso o objeto passado seja nulo (isto é, ausente), ele gera `false`.

Em seguida, o método verifica se os objetos `Class` são iguais (isto é, o objeto passado precisa ser um objeto `Person`). Se forem iguais, cada valor de atributo do objeto passado é verificado para ver se corresponde ao valor por valor com o estado da instância `Person` fornecida. Caso os valores de atributo sejam nulos, `equals()` verifica todos os que conseguir; havendo correspondência, os objetos são considerados iguais. Talvez você não queira esse comportamento para todos os programas, mas funciona para a maioria dos propósitos.

Exercício: Gerar um `equals()` para `Employee`

Tente seguir as etapas de "Geração automática de `equals()`" para gerar um `equals()` para `Employee`. Depois de ter o `equals()` gerado, inclua o caso de teste JUnit a seguir (que chamei de `yetAnotherTest()`) nele:

```
@Test
public void yetAnotherTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
  Employee employee1 = new Employee();
  employee1.setName("J Smith");
  Employee employee2 = new Employee();
  employee2.setName("J Smith");
  l.info("Q: employee1 == employee2?      A: " + (employee1 == employee2));
  l.info("Q: employee1.equals(employee2)? A: " + employee1.equals(employee2));
}
```

Mostrar mais

Se executar o código, deve ver a saída a seguir:

```
Sep 19, 2015 11:27:23 AM com.makotojava.intro.EmployeeTest yetAnotherTest
INFO: Q: employee1 == employee2?      A: false
Sep 19, 2015 11:27:23 AM com.makotojava.intro.EmployeeTest yetAnotherTest
INFO: Q: employee1.equals(employee2)? A: true
```

Mostrar mais

Neste caso, uma correspondência apenas em `Name` foi suficiente para convencer `equals()` de que os dois objetos são iguais. Tente incluir mais atributos neste exemplo e veja o que será obtido.

Exercício: Substituir `toString()`

Lembra-se do método `printAudit()` do início desta seção? Se achou que ele estava trabalhando demais, você acertou. Formatar o estado de um objeto em `String` é um padrão tão comum que os designers da linguagem Java o inseriram no próprio `Object`, em um método chamado (evidentemente) `toString()`. A implementação padrão de `toString()` não é muito útil, mas todo objeto tem um. Neste exercício, você torna o `toString()` padrão um pouco mais útil.

Se acha que o Eclipse pode gerar um método `toString()` para você, tem razão. Volte para seu Project Explorer e clique com o botão direito na classe `Person`; depois, escolha **Source > Generate toString()...**. Você verá uma caixa de diálogo semelhante a da Caixa de diálogo Generate hashCode() e equals(). Selecione todos os atributos e clique em **OK**. Faça o mesmo para `Employee`. O código gerado pelo Eclipse para `Employee` é mostrado na Listagem 9.

Um método `toString()` gerado pelo Eclipse

```
@Override
public String toString() {
  return "Employee [taxpayerIdentificationNumber=" + taxpayerIdentificationNumber + ",
      employeeNumber=" + employeeNumber + ", salary=" + salary + "]";
}
```

Mostrar mais

O código gerado pelo Eclipse para `toString` não inclui a superclasse com seu `toString()` (`Employee` como superclasse sendo `Person`). É possível corrigir isso rapidamente, usando o Eclipse, com esta substituição:

```
@Override
public String toString() {
  return super.toString() + "Employee [taxpayerIdentificationNumber=" + taxpayerIdentification
    ", employeeNumber=" + employeeNumber + ", salary=" + salary + "]";
}
```

Mostrar mais

A adição de `toString()` deixa `printAudit()` muito mais simples:

```
@Override
  public void printAudit(StringBuilder buffer) {
  buffer.append(toString());
}
```

Mostrar mais

`toString()` passou a fazer o trabalho pesado de formatar o estado atual do objeto; você precisa apenas colocar o que é gerado em `StringBuilder` e retornar.

Recomendo sempre implementar `toString()` nas suas classes, mesmo se for apenas para fins de suporte. É praticamente inevitável que, em algum momento, você queira ver qual é o estado de um objeto enquanto seu aplicativo está em execução e `toString()` é uma ótima oportunidade para fazer isso.

Exceções

Nenhum programa funciona o tempo todo e os designers da linguagem Java sabiam disso. Nesta seção, aprenda sobre os mecanismos integrados da plataforma Java para lidar com situações em que seu código não funcionar exatamente como o planejado.

Princípios básicos da manipulação de exceção

Uma *exceção* é um evento que ocorre durante a execução do programa e interrompe o fluxo normal das suas instruções. A manipulação de exceção é uma técnica essencial da programação de Java. Em essência, você agrupa seu código em um bloco `try` (que significa “experimente isto e diga-me se causa uma exceção”) e o utiliza para `capturar` vários tipos de exceções.

Para começar com a manipulação de exceção, dê uma olhada no código na Listagem 10.

Está vendo o erro?

```
@Test
public void yetAnotherTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
//    Employee employee1 = new Employee();
  Employee employee1 = null;
  employee1.setName("J Smith");
  Employee employee2 = new Employee();
  employee2.setName("J Smith");
  l.info("Q: employee1 == employee2?      A: " + (employee1 == employee2));
  l.info("Q: employee1.equals(employee2)? A: " + employee1.equals(employee2));
}
```

Mostrar mais

Observe que a referência `Employee` foi configurada como `null`. Execute este código e obtenha a saída a seguir:

```
java.lang.NullPointerException
  at com.makotojava.intro.EmployeeTest.yetAnotherTest(EmployeeTest.java:49)
  .
  .
  .
```

Mostrar mais

Essa saída diz que você está tentando fazer referência a um objeto por meio de uma referência (ponteiro) `null`, o que é um erro de desenvolvimento muito grave.

Felizmente, é possível usar os blocos `try` e `catch` para capturá-lo (juntamente com uma ajuda de `finally`).

Aviso do IDE

Enquanto trabalhava com o código da Listagem 10 , provavelmente percebeu que o Eclipse avisa do erro em potencial com a mensagem: `Null pointer access: The variable employee1`

`try`, `catch` e `finally`

A Listagem 11 mostra o código de erro da Está vendo o erro? limpa com os blocos de códigos padrão para a manipulação de exceção: `try`, `catch` e `finally`.

`can only be null at this location`.
O Eclipse avisa você sobre muitos erros em potencial no desenvolvimento mais uma vantagem de utilizar um IDE para fazer desenvolvimento de Java.

Capturando uma exceção

```
@Test
public void yetAnotherTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
  //    Employee employee1 = new Employee();
  try {
    Employee employee1 = null;
    employee1.setName("J Smith");
    Employee employee2 = new Employee();
    employee2.setName("J Smith");
    l.info("Q: employee1 == employee2?      A: " + (employee1 == em
    l.info("Q: employee1.equals(employee2)? A: " + employee1.equals
  } catch (Exception e) {
    l.severe("Caught exception: " + e.getMessage());
  } finally {
    // Always executes
```

Mostrar mais

Juntos, os blocos `try`, `catch` e `finally` formam uma rede para capturar exceções. Em primeiro lugar, a instrução `try` agrupa um código que pode lançar uma exceção. Neste caso, a execução cai imediatamente no bloco `catch` ou no *manipulador de exceções*. Quando todas as tentativas e capturas são concluídas, a execução continua para o bloco `finally`, independentemente de uma exceção ter sido lançada. Quando capturar uma exceção, é possível tentar se recuperar graciosamente dela ou sair do programa (ou método).

Na Listagem 11, o programa se recupera do erro e, em seguida, imprime a mensagem da exceção:

```
Sep 19, 2015 2:01:22 PM com.makotojava.intro.EmployeeTest yetAnotherTest
SEVERE: Caught exception: null
```

Mostrar mais

A hierarquia de exceções

A linguagem Java incorpora toda uma hierarquia de exceções composta por muitos tipos de exceções agrupadas em duas categorias principais:

- **As exceções verificadas** são verificadas pelo compilador (o que significa que o compilador garante que sejam manipuladas em algum lugar do seu código).
- **Exceções não verificadas** (também chamadas de *exceções de tempo de execução*) não são verificadas pelo compilador.

Quando um programa causa uma exceção, diz-se que ele *lança* a exceção. Uma exceção verificada é declarada para o compilador por qualquer método com a palavra-chave `throws` na sua assinatura de método. Em seguida, vem uma lista separada por vírgula de exceções que o método poderia potencialmente lançar durante sua execução. Se seu código chamar um método que especifica o lançamento de um ou mais tipos de exceções, é necessário lidar com ele de alguma forma ou incluir `throws` na sua assinatura de método para passar esse tipo de exceção adiante.

Quando ocorre uma exceção, o Java Runtime procura um manipulador de exceções em algum lugar da pilha. Se não encontrar um até o momento de chegar ao topo da pilha, interrompe o programa abruptamente, como foi visto na Está vendo o erro?.

Diversos blocos `catch`

É possível ter diversos blocos `catch`, mas eles precisam ser estruturados de maneira específica. Se quaisquer exceções forem subclasses de outras exceções, as classes-filhas são colocadas na frente das classes-pais na ordem dos blocos `catch`. A Listagem 12 mostra um exemplo de tipos de exceções diferentes estruturadas em sua sequência hierárquica correta.

Exemplo de hierarquia de exceções

```
@Test
public void exceptionTest() {
  Logger l = Logger.getLogger(Employee.class.getName());
  File file = new File("file.txt");
  BufferedReader bufferedReader = null;
  try {
    bufferedReader = new BufferedReader(new FileReader(file));
    String line = bufferedReader.readLine();
    while (line != null) {
      // Read the file
    }
  } catch (FileNotFoundException e) {
    l.severe(e.getMessage());
  } catch (IOException e) {
    l.severe(e.getMessage());
```

Mostrar mais

Neste exemplo, `FileNotFoundException` é uma classe-filha de `IOException`; portanto, precisa ficar na frente do bloco `IOException catch`. Além disso, `IOException` é uma classe-filha de `Exception` e precisa ficar na frente do bloco `Exception catch`.

`try-with-resources` como blocos

A partir do JDK 7, o gerenciamento de recurso ficou muito mais simples. À medida que trabalha mais com arquivos e o pacote `java.io`, entenderá a nova sintaxe. O código na Exemplo de hierarquia de exceções precisou declarar uma variável para conter a referência `bufferedReader` e, em `finally`, precisou fechar `BufferedReader`.

A sintaxe `try-with-resources` mais recente fecha automaticamente recursos quando o bloco `try` sai do escopo. A Listagem 13 mostra a sintaxe mais compacta.

Sintaxe Resource-management

```
@Test
public void exceptionTestTryWithResources() {
  Logger l = Logger.getLogger(Employee.class.getName());
  File file = new File("file.txt");
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader(file))) {
      String line = bufferedReader.readLine();
      while (line != null) {
// Read the file
      }
    } catch (Exception e) {
```

```
        l.severe(e.getMessage());
      }
}
```

Mostrar mais

Essencialmente, variáveis de recurso são designadas depois de `try` entre parênteses; quando o bloco `try` sai do escopo, esses recursos são fechados automaticamente. Os recursos precisam implementar a interface `java.lang.AutoCloseable`; se você tentar usar essa sintaxe em uma classe de recurso que não faz isso, será avisado pelo Eclipse.

Construindo aplicativos Java

Nesta seção, você continua construindo `Person` como um aplicativo Java. No caminho, é possível entender melhor como um objeto, ou coleção de objetos, evolui e se torna um aplicativo.

Elementos de um aplicativo Java

Todos os aplicativos Java precisam de um ponto de entrada em que o Java Runtime saiba que deve começar a executar o código. Esse ponto de entrada é `main()`. Os objetos do domínio normalmente não têm métodos `main()`, mas pelo menos uma classe de cada aplicativo deve ter.

Você está trabalhando desde a Parte 1 com o exemplo de um aplicativo de recursos humanos que inclui `Person` e suas subclasses `Employee`. Agora, é possível ver o que acontece quando uma nova classe é incluída no aplicativo.

Criando uma classe do driver

O propósito de uma *classe do driver* (como o nome sugere) é “conduzir” um aplicativo. Observe que esse driver simples para o aplicativo de recursos humanos contém um `main()`:

```
package com.makotojava.intro;
public class HumanResourcesApplication {
  public static void main(String[] args) {
  }
}
```

Mostrar mais

Crie uma classe do driver no Eclipse usando o mesmo procedimento que você usou para criar `Person` e `Employee`. Nomeie a classe como `HumanResourcesApplication`, lembrando-se de selecionar a opção para incluir um método `main()` na classe. O Eclipse gerará a classe para você.

Inclua algum código no seu novo método `main()` para que seja semelhante ao seguinte:

```
package com.makotojava.intro;
import java.util.logging.Logger;

public class HumanResourcesApplication {
  private static final Logger log = Logger.getLogger(HumanResourcesApplication.class.getName()
  public static void main(String[] args) {
```

```
    Employee e = new Employee();
    e.setName("J Smith");
    e.setEmployeeNumber("0001");
    e.setTaxpayerIdentificationNumber("123-45-6789");
    e.setSalary(BigDecimal.valueOf(45000.0));
    e.printAudit(log);
  }
}
```

Mostrar mais

Agora, lance a classe `HumanResourcesApplication` e acompanhe sua execução. Você deve ver esta saída (com a barra invertida indicando uma continuação de linha):

```
Sep 19, 2015 7:59:37 PM com.makotojava.intro.Person printAudit
INFO: Name=J Smith,Age=0,Height=0,Weight=0,EyeColor=null,Gender=null\
TaxpayerIdentificationNumber=123-45-6789,EmployeeNumber=0001,Salary=45000.00
```

Mostrar mais

Isso basta para criar um aplicativo Java simples. Na próxima seção, você começará a ver algumas sintaxes e bibliotecas que podem ajudá-lo a desenvolver aplicativos mais complexos.

Herança

Você já encontrou exemplos de herança algumas vezes neste tutorial. Esta seção revisa alguns materiais da Parte 1 sobre herança e explica em mais detalhes como ela funciona — incluindo a hierarquia de herança, construtores e abstração de herança.

Como a herança funciona

Existem classes em código Java em hierarquias. As classes acima de determinada classe em uma hierarquia são *superclasses* dessa classe. Essa classe específica é uma *subclasse* de cada classe superior na hierarquia. Uma subclasse herda de suas superclasses. A classe `java.lang.Object` fica na parte superior da hierarquia de classes, o que significa que cada classe Java é uma subclasse de, e herda de, `Object`.

Suponha, por exemplo, que você tem uma classe `Person` que se parece com aquela da Listagem 14.

Classe pública `Person`

```
public class Person {

  public static final String STATE_DELIMITER = "~";

  public Person() {
    // Default constructor
  }

  public enum Gender {
    MALE,
    FEMALE,
    UNKNOWN
  }

  public Person(String name, int age, int height, int weight, String eyeColor, G
```

Mostrar mais

A classe `Person` na Listagem 14 herda implicitamente de `Object`. Como herdar de `Object` é algo presumido para cada classe, não é necessário digitar `extends Object` para cada classe definida por você. Porém, o que significa dizer que uma classe herda da sua superclasse? Significa, simplesmente, que `Person` tem acesso às variáveis e métodos expostos em suas superclasses. Neste caso, `Person` pode ver e usar os métodos e variáveis públicos de `Object`, assim como os métodos e variáveis protegidos de `Object`.

Definindo uma hierarquia de classes

Agora, suponha que você possui uma classe `Employee` que herda de `Person`. `Employee` como definição de classe (ou *gráfico de herança*) seria semelhante ao seguinte:

```
public class Employee extends Person {

  private String taxpayerIdentificationNumber;
  private String employeeNumber;
  private BigDecimal salary;
  // . . .
}
```

Mostrar mais

O `Employee` como gráfico de herança implica que `Employee` tem acesso a todas as variáveis e métodos públicos e privados em `Person` (porque `Employee` amplia diretamente `Person`), assim como em `Object` (porque `Employee` na verdade amplia `Object` também, embora indiretamente). Contudo, uma vez que `Employee` e `Person` estão no mesmo pacote, `Employee` também tem acesso às variáveis e métodos de *pacote privado* (às vezes chamados de *fáceis de usar*) em `Person`.

Para ir mais a fundo na hierarquia de classes, é possível criar uma terceira classe que amplia `Employee`:

```
public class Manager extends Employee {
  // . . .
}
```

Mostrar mais

Na linguagem Java, qualquer classe pode ter no máximo uma superclasse, mas uma classe pode ter qualquer número de subclasses. Trata-se da coisa mais importante para lembrar sobre hierarquia de herança na linguagem Java.

Diversas heranças versus herança única

Linguagens como C++ fornecem suporte ao conceito de *diversas heranças*: Em qualquer ponto da hierarquia, uma classe pode herdar de uma ou mais classes. A linguagem Java fornece uma única classe. Assim, a hierarquia de classes para qualquer classe Java sempre consiste de uma linha reta até chegar a `java.lang.Object`. No entanto, como você aprenderá mais tarde no tutorial, a linguagem Java fornece suporte para a implementação de diversas interfaces em uma única classe, oferecendo um tipo de solução alternativa para a herança única.

Construtores e herança

Por não serem membros orientados a objetos completamente desenvolvidos, os construtores não são herdados; em vez disso, você precisa implementá-los explicitamente em subclasses. Antes de tratar desse assunto, revisarei algumas regras básicas sobre como os construtores são definidos e chamados.

Princípios básicos dos construtores

Lembre-se que um construtor sempre tem o mesmo nome que a classe que foi utilizado para construir, mas não possui um tipo de retorno. Por exemplo:

```
public class Person {
  public Person() {
  }
}
```

Mostrar mais

Cada classe tem pelo menos um construtor; se não definir explicitamente um construtor para sua classe, o compilador gerará um para você, chamado de *construtor padrão*. A definição de classe precedente e esta aqui são idênticas em termos de função:

```
public class Person {
}
```

Mostrar mais

Chamando um construtor de superclasse

Para chamar um construtor de superclasse à exceção do construtor padrão, é necessário agir explicitamente. Suponha, por exemplo, que `Person` possui um construtor que pega somente o nome do objeto `Person` que está sendo criado. A partir do construtor padrão de `Employee`, você poderia chamar o construtor `Person` mostrado na Listagem 15:

Inicializando um novo `Employee`

```
public class Person {
  private String name;
  public Person() {
  }
  public Person(String name) {
    this.name = name;
  }
}

// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee() {
    super("Elmer J Fudd");
  }
}
```

Mostrar mais

Provavelmente, porém, você nunca desejará inicializar um novo objeto `Employee` dessa maneira. Até ficar mais familiarizado com conceitos orientados a objetos, e com a sintaxe Java em geral, sugerimos implementar construtores de superclasse em subclasses caso ache que irá precisar deles. A Listagem 16 define um construtor em `Employee` que se parece com aquele de `Person` para que haja correspondência. Essa abordagem é muito menos confusa do ponto de vista da manutenção.

Chamando uma superclasse

```
public class Person {
  private String name;
  public Person(String name) {
    this.name = name;
  }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee(String name) {
    super(name);
  }
}
```

Mostrar mais

Declarando um construtor

A primeira coisa que um construtor faz é chamar o construtor padrão de sua superclasse imediata, a menos que você — na primeira linha de código do construtor — chame um construtor diferente. Por exemplo, estas duas declarações são funcionalmente idênticas, então escolha uma:

```
public class Person {
  public Person() {
  }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee() {
  }
}
```

Mostrar mais

Ou:

```
public class Person {
  public Person() {
  }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee() {
  super();
  }
}
```

Mostrar mais

Construtores no-arg

Se você oferecer um construtor alternativo, precisa fornecer explicitamente o construtor padrão; caso contrário, ele fica indisponível. Por exemplo, o código a seguir apresenta um erro de compilação:

```
public class Person {
  private String name;
  public Person(String name) {
```

```
    this.name = name;
  }
}
// Meanwhile, in Employee.java
public class Employee extends Person {
  public Employee() {
  }
}
```

Mostrar mais

A classe `Person` neste exemplo não tem um construtor padrão, porque oferece um construtor alternativo sem incluir explicitamente o construtor padrão.

Como os construtores chamam construtores

Um construtor de dentro de uma classe pode ser chamado por outro construtor por meio da palavra-chave `this`, juntamente com uma lista de argumentos. Como em `super()`, a chamada `this()` precisa ser a primeira linha no construtor. Por exemplo:

```
public class Person {
  private String name;
  public Person() {
    this("Some reasonable default?");
  }
  public Person(String name) {
    this.name = name;
  }
}
```

Mostrar mais

Você vê esse idioma frequentemente, em que um construtor delega para outro, passando um valor padrão se tal construtor for chamado. Também é uma ótima maneira de incluir um novo construtor em uma classe enquanto miminiza o impacto em um código que já usa um construtor mais antigo.

Níveis de acesso do construtor

Os construtores podem ter qualquer nível de acesso que você quiser e algumas regras de visibilidade se aplicam. A Tabela 1 resume as regras de acesso do construtor.

Regras de acesso do construtor

Modificador de acesso do construtor	**Descrição**
`public`	O construtor pode ser chamado por qualquer classe.
`protected`	O construtor pode ser chamado por uma classe no mesmo pacote ou por qualquer subclasse.
Nenhum modificador (*pacote privado*)	O construtor pode ser chamado por qualquer classe no mesmo pacote.
`private`	O construtor pode ser chamado apenas pela classe em que foi definido.

Talvez você consiga pensar em casos de uso em que os construtores seriam declarados como `protected` ou até mesmo de pacote privado, mas como um construtor do tipo `private` pode ser útil? Utilizei construtores privados quando não quis permitir a criação direta de um objeto por meio da palavra-chave `new` ao implementar, por exemplo, o padrão de fábrica . Neste caso, eu usaria um método estático para criar instâncias da classe; esse método — por estar incluído na classe — estaria autorizado a chamar o construtor privado.

Herança e abstração

Se uma subclasse substituir um método de uma superclasse, o método ficará essencialmente oculto porque chamá-lo por meio de uma referência à subclasse chama a versão do método da subclasse, não a versão da superclasse. Isso não quer dizer que o método de superclasse deixou de ser acessível. A subclasse pode chamar o método de superclasse ao criar um prefácio para o nome do método com a palavra-chave `super` (e, ao contrário das regras do construtor, isso pode ser feito a partir de qualquer linha no método de subclasse ou até mesmo em um método totalmente diferente). Por padrão, um programa Java chama o método de subclasse se for chamado por meio de uma referência à subclasse.

O mesmo se aplica a variáveis, desde que o responsável pela chamada tenha acesso à variável (ou seja, a variável está visível para o código que tenta acessá-la). Esse detalhe pode causar muitas preocupações enquanto você adquire proficiência na programação de Java. O Eclipse fornece muitos avisos de que você está ocultando uma variável de uma superclasse ou que uma chamada de método não chamará aquilo que você espera.

Em um contexto OOP, *abstração* se refere a generalizar dados e o comportamento até um tipo mais alto na hierarquia de herança do que a classe atual. Quando variáveis ou métodos são movidos de uma subclasse para uma superclasse, dizemos que você está *abstraindo* esses membros. O principal motivo para fazer isso é reutilizar um código comum ao colocá-lo o mais alto possível na hierarquia. Colocar o código comum em um local facilita a manutenção.

Classes e métodos abstratos

Às vezes, você deseja criar classes que funcionam apenas como abstrações e não precisam, necessariamente, ser instanciadas sempre. Tais classes são chamadas de *classes abstratas*. Pelo mesmo token, há momentos em que determinados métodos precisam ser implementados de forma diferente para cada subclasse que implementa a superclasse. Esses métodos são *métodos abstratos*. Estas são algumas regras básicas para classes e métodos abstratos:

- Qualquer classe pode ser declarada como `abstrata`.
- As classes abstratas não podem ser instanciadas.
- Um método abstrato não pode conter um corpo de método.
- Qualquer classe com um método abstrato precisa ser declarada como `abstrata`.

Usando a abstração

Suponha que você não quer permitir que a classe `Employee` seja instanciada diretamente. Basta declará-la usando a palavra-chave `abstract` e pronto:

```
public abstract class Employee extends Person {
  // etc.
}
```

Mostrar mais

Caso tente executar esse código, você obterá um erro de compilação:

```
public void someMethodSomwhere() {
  Employee p = new Employee();// compile error!!
}
```

Mostrar mais

O compilador está reclamando que `Employee` é abstrato e não pode ser instanciado.

O poder da abstração

Suponha que você precisa de um método para examinar o estado de um objeto `Employee` e garantir que seja válido. Essa necessidade pareceria ser comum para todos os objetos `Employee`, mas se comportaria de forma suficientemente diferente entre todas as possíveis subclasses nas quais tem potencial zero de reutilização. Neste caso, você declara o método `validate()` como `abstract` (forçando todas as subclasses a implementá-lo):

```
public abstract class Employee extends Person {
  public abstract boolean validate();
}
```

Mostrar mais

Cada subclasse direta de `Employee` (como `Manager`) passou a ser obrigada a implementar `validate()`. No entanto, após uma subclasse ter implementado o método `validate()`, nenhuma de suas subclasses precisa implementá-lo.

Suponha, por exemplo, que você tem um objeto `Executive` que amplia `Manager`. Essa definição seria válida:

```
public class Executive extends Manager {
  public Executive() {
  }
}
```

Mostrar mais

Quando (não) abstrair: Duas regras

A primeira regra básica é não abstrair no design inicial. O uso de classes abstratas no início do design força você a percorrer um caminho específico, o que poderia restringir seu aplicativo. Lembre-se: um comportamento comum (que é a finalidade de ter classes abstratas) pode ser sempre refatorado mais

acima no gráfico de herança. Quase sempre é melhor refatorar depois de descobrir que precisa fazer isso. O Eclipse tem um suporte excelente para a refatoração.

Em segundo lugar, por mais eficientes que as classes abstratas sejam, resista a usá-las. A menos que suas superclasses contenham muitos comportamentos comuns e não sejam significativas por conta própria, deixe que permaneçam não abstratas. Gráficos de herança profundos podem dificultar a manutenção do código. Considere a troca entre classes que são grandes demais e um código passível de manutenção.

Designações: Classes

É possível designar uma referência de uma classe para uma variável de um tipo que pertence à outra classe, mas algumas regras se aplicam. Veja este exemplo:

```
Manager m = new Manager();
Employee e = new Employee();
Person p = m; // okay
p = e; // still okay
Employee e2 = e; // yep, okay
e = m; // still okay
e2 = p; // wrong!
```

Mostrar mais

A variável de destino precisa ser de um supertipo de classe que pertence à referência de origem ou o compilador fornecerá um erro. Basicamente, tudo o que está no lado direito da designação precisa ser uma subclasse ou a mesma classe daquilo que está à esquerda. Caso contrário, é possível que designações de objetos com diferentes gráficos de herança (tais como `Manager` e `Employee`) sejam designadas para uma variável do tipo errado. Agora considere este exemplo:

```
Manager m = new Manager();
Person p = m; // so far so good
Employee e = m; // okay
Employee e = p; // wrong!
```

Mostrar mais

Embora um `Employee` seja `Person`, definitivamente não é um `Manager` e o compilador aplica essa distinção.

Interfaces

Nesta seção, você começa a aprender sobre interfaces e começa a usá-las no seu código Java.

Definindo uma interface

Uma *interface* é um conjunto nomeado de comportamentos (ou elementos de dados constantes) para o qual um implementador precisa fornecer código. Uma interface especifica o comportamento que a implementação fornece, mas não como é realizada.

Definir uma interface é algo simples:

```
public interfaceinterfaceName {
    returnType methodName(argumentList);
  }
```

Mostrar mais

Uma declaração de interface se parece com uma declaração de classe, exceto pelo fato de que você usa a palavra-chave `interface`. É possível nomear a interface como você quiser (sujeito às regras do idioma), mas, por convenção, os nomes das interfaces se parecem com nomes de classe.

Os métodos definidos em uma interface não possuem corpo de método. O implementador da interface é responsável por fornecer o corpo de método (tal como com métodos abstratos).

Você define as hierarquias de interfaces, assim como faz para as classes, exceto pelo fato de que uma classe única pode implementar quantas interfaces quiser. (Lembre-se: uma classe pode ampliar apenas uma classe.) Se uma classe ampliar outra e implementar uma interface ou interfaces, as interfaces são listadas após a classe ampliada, deste modo:

```
public class Manager extends Employee implements BonusEligible, StockOptionRecipient {
  // Etc...
}
```

Mostrar mais

Interfaces de marcador

Uma interface não precisa ter nenhum corpo. A definição a seguir é perfeitamente aceitável, por exemplo:

```
public interface BonusEligible {
}
```

Mostrar mais

De modo geral, tais interfaces são chamadas de *interfaces de marcador*, porque marcam uma classe como a implementação dessa interface, mas não oferecem nenhum comportamento explícito especial.

Depois de saber tudo isso, realmente definir uma interface é fácil:

```
public interface StockOptionRecipient {
  void processStockOptions(int numberOfOptions, BigDecimal price);
}
```

Mostrar mais

Implementando interfaces

Para usar uma interface, você a *implementa*, o que significa que fornece um corpo de método que, por sua vez, fornece o comportamento para cumprir o contrato da interface. Você usa a criação `implements` como palavra-chave para implementar uma interface:

```
public class
className
extends
superclassName
implements
interfaceName {
  // Class Body
}
```

Mostrar mais

Suponha que você implementou a interface `StockOptionRecipient` na classe `Manager`, como mostrado na Listagem 17:

Implementando uma interface

```
public class Manager extends Employee implements StockOptionRecipient {
  public Manager() {
  }
  public void processStockOptions (int numberOfOptions, BigDecimal price) {
    log.info("I can't believe I got " + number + " options at $" +
    price.toPlainString() + "!");
  }
}
```

Mostrar mais

Quando implementa a interface, você fornece um comportamento para o método ou métodos contidos na interface. É preciso implementar os métodos com assinaturas que correspondem às assinaturas da interface, com a adição do modificador de acesso `public`.

Gerando interfaces no Eclipse

O Eclipse pode gerar facilmente a assinatura de método correta para você caso você decida que uma das suas classes deve implementar uma interface. Basta mudar a assinatura de classe para implementar a interface. O Eclipse coloca uma linha ondulada vermelha sob a classe, sinalizando-a como estando em erro porque a classe não fornece os métodos na interface. Clique no nome de classe, pressione Ctrl + 1 e o Eclipse sugere “correções rápidas” para você. Dentre elas, escolha **Add Unimplemented Methods** e o Eclipse gera os métodos para você, colocando-os na parte inferior do arquivo de origem.

Uma classe abstrata pode declarar que implementa uma interface específica, mas você não precisa implementar todos os métodos nela. As classes abstratas não precisam fornecer implementações para todos os métodos que afirmam implementar. Contudo, a primeira classe concreta (ou seja, a primeira que pode ser instanciada) precisa implementar todos os métodos que a hierarquia não implementa.

Usando interfaces

Uma interface define um novo tipo de dados de *referência*, que pode ser utilizado para se referir a uma interface em qualquer lugar em que você faria referência a uma classe. Essa capacidade inclui o momento de declarar uma variável de referência, ou lançar de um tipo para outro, tal como mostrado na Listagem 18.

Designando uma nova instância `Manager` para uma referência `StockOptionEligible`

```
package com.makotojava.intro;
import java.math.BigDecimal;
import org.junit.Test;
public class ManagerTest {
  @Test
  public void testCalculateAndAwardStockOptions() {
    StockOptionEligible soe = new Manager();// perfectly valid
    calculateAndAwardStockOptions(soe);
    calculateAndAwardStockOptions(new Manager());// works too
    }
    public static void calculateAndAwardStockOptions(StockOptionEligible soe) {
    BigDecimal reallyCheapPrice = BigDecimal.valueOf(0.01);
    int numberOfOptions = 10000;
    soe.awardStockOptions(numberOfOptions, reallyCheapPrice);
  }
```

Mostrar mais

Como é possível ver, é válido designar uma nova instância `Manager` para uma referência `StockOptionEligible` e passar uma nova instância `Manager` para um método que espera uma referência `StockOptionEligible`.

Designações: Interfaces

É possível designar uma referência de uma classe que implementa uma interface com uma variável de um tipo de interface, mas algumas regras se aplicam. Na Designando uma nova instância `Manager` para uma referência `StockOptionEligible`, é possível ver que designar uma instância `Manager` para uma referência variável `StockOptionEligible` é válido. O motivo é que a classe `Manager` implementa tal interface. Entretanto, tal designação não seria válida:

```
Manager m = new Manager();
  StockOptionEligible soe = m; //okay
  Employee e = soe; // Wrong!
```

Mostrar mais

Como `Employee` é um supertipo de `Manager`, esse código poderia parecer adequado, a princípio, mas não é. Como `Manager` é uma especialização de `Employee`, é diferente e, neste caso específico, implementa uma interface que `Employee` não implementa.

Designações como essas seguem as regras de designação vistas na seção "Herança". Tal como acontece com as classes, é possível designar uma referência de interface a uma variável do mesmo tipo ou um tipo de superinterface.

Classes aninhadas

Nesta seção, você aprende sobre classes aninhadas e onde e como usá-las.

Onde usar classes aninhadas

Como o nome sugere, uma *classe aninhada* é definida dentro de outra classe. Esta é uma classe aninhada:

```
public class EnclosingClass {
  . . .
  public class NestedClass {
  . . .

  }
}
```

Mostrar mais

Como variáveis e métodos de membro, as classes Java também podem ser definidas em qualquer escopo, incluindo `public`, `private` ou `protected`. As classes aninhadas podem ser úteis quando você deseja lidar com o processamento interno dentro da sua classe e maneira orientada a objetos, mas tal funcionalidade está limitada à classe em que é necessária.

Normalmente, você utiliza uma classe aninhada quando precisa de uma classe fortemente associada à classe em que é definida. Uma classe aninhada tem acesso aos dados privados dentro da classe delimitadora, mas tal estrutura traz consigo alguns efeitos colaterais que não são óbvios quando você começa a trabalhar com classes aninhadas (ou internas).

Escopo em classes aninhadas

Como possui escopo, uma classe aninhada é limitada pelas regras de escopo. Por exemplo, uma variável de membro pode ser acessada apenas por meio de uma instância da classe (um objeto). O mesmo se aplica a uma classe aninhada.

Suponha que há o relacionamento a seguir entre um `Manager` e uma classe aninhada chamada `DirectReports`, que é uma coleção de `Employees` subordinados ao `Manager`:

```
public class Manager extends Employee {
  private DirectReports directReports;
  public Manager() {
  this.directReports = new DirectReports();
  }
  . . .
  private class DirectReports {
  . . .
  }
}
```

Mostrar mais

Assim como cada objeto `Manager` representa um único ser humano, o objeto `DirectReports` representa uma coleção de pessoas reais (funcionários) subordinados a um gerente. `DirectReports` variam conforme o `Manager`. Neste caso, faz sentido que eu me refira somente à classe aninhada `DirectReports` no contexto da instância delimitadora de `Manager`; portanto, tornei-a do tipo `private`.

Classes públicas aninhadas

Por ser do tipo `private`, apenas `Manager` pode criar uma instância de `DirectReports`. Suponha, porém, que você queria dar a uma entidade externa a capacidade de criar instâncias de `DirectReports`. Neste caso, parece que você poderia dar à classe `DirectReports` um escopo do tipo `public`; assim, qualquer código externo poderia criar instâncias de `DirectReports`, tal como mostrado na Listagem 19.

Criando instâncias de `DirectReports` instances: First attempt

```
public class Manager extends Employee {
  public Manager() {
  }
  . . .
  public class DirectReports {
  . . .
  }
}
//
public static void main(String[] args) {
  Manager.DirectReports dr = new Manager.DirectReports();// This won't work!
}
```

Mostrar mais

O código na Criando instâncias de `DirectReports` instances: First attempt não funciona e você provavelmente está se perguntando o porquê. O problema (bem como sua solução) está na forma como `DirectReports` é definido dentro de `Manager` e com as regras do escopo.

As regras do escopo, revistas

Se tivesse uma variável de membro de `Manager`, você esperaria que o compilador exigisse que fosse feita uma referência a um objeto `Manager` antes que fosse possível referenciá-lo, certo? Bem, o mesmo se aplica a `DirectReports`, pelo menos do modo como você o definiu na Criando instâncias de `DirectReports` instances: First attempt.

Para criar uma instância de uma classe pública aninhada, você usa uma versão especial do operador `new`. Combinado com uma referência a uma instância delimitadora de uma classe externa, `new` permite que você crie uma instância da classe aninhada:

```
public class Manager extends Employee {
  public Manager() {
  }
  . . .
  public class DirectReports {
  . . .
  }
  }
// Meanwhile, in another method somewhere...
public static void main(String[] args) {
  Manager manager = new Manager();
  Manager.DirectReports dr = manager.new DirectReports();
}
```

Mostrar mais

Observe que a sintaxe pede uma referência à instância delimitadora, além de um ponto e da palavra-chave new, seguida pela classe que você deseja criar.

Classes estáticas internas

Às vezes, você deseja criar uma classe que esteja fortemente ligada (conceitualmente) a uma classe, mas em que as regras de escopo sejam um pouco mais flexíveis, sem exigir uma referência a uma instância delimitadora. É aí que entram as classes estáticas internas. Um exemplo comum é implementar um `Comparator`, que é utilizado para comparar duas instâncias da mesma classe, geralmente para fins de ordenar (ou classificar) as classes:

```
public class Manager extends Employee {
  . . .
  public static class ManagerComparator implements Comparator<Manager> {
  . . .
  }
  }
// Meanwhile, in another method somewhere...
public static void main(String[] args) {
  Manager.ManagerComparator mc = new Manager.ManagerComparator();
  . . .
}
```

Mostrar mais

Neste caso, você não precisa de uma instância delimitadora. As classes estáticas internas agem como as contrapartes de classe Java regulares e devem ser utilizadas apenas quando você precisa ligar uma classe fortemente com sua definição. Claramente, no caso de uma classe do utilitário como `ManagerComparator`, criar uma classe externa é desnecessário e possivelmente atravanca seu código base. Definir tais classes como classes estáticas internas é o caminho certo a seguir.

Classes anônimas internas

Com a linguagem Java, é possível declarar classes praticamente em qualquer lugar, inclusive no meio de um método, se necessário, e mesmo sem fornecer um nome para a classe. Essa capacidade é basicamente um truque do compilador, mas há ocasiões em que é útil ter classes anônimas internas.

A Listagem 20 utiliza o exemplo da Implementando uma interface, incluindo um método padrão para manipular tipos de `Employee` que não são `StockOptionEligible`. A listagem começa com um método em `HumanResourcesApplication` para processar as opções de ação, seguido por um teste JUnit para orientar o método:

Manipulando tipos de `Employee` que não são `StockOptionEligible`

```
// From HumanResourcesApplication.java
public void handleStockOptions(final Person person, StockOptionProcessingCallback callback) {
  if (person instanceof StockOptionEligible) {
    // Eligible Person, invoke the callback straight up
    callback.process((StockOptionEligible)person);
  } else if (person instanceof Employee) {
    // Not eligible, but still an Employee. Let's cobble up a
    /// anonymous inner class implementation for this
    callback.process(new StockOptionEligible() {
      @Override
      public void awardStockOptions(int number, BigDecimal price) {
```

```
        // This employee is not eligible
        log.warning("It would be nice to award " + number + " of shares at $" +
            price.setScale(2, RoundingMode.HALF_UP).toPlainString() +
            "  but unfortunately  Employee " + person.getName() +
```

Mostrar mais

Neste exemplo, forneço implementações de duas interfaces que usam classes anônimas internas. Primeiramente, existem duas implementações separadas de `StockOptionEligible` — uma para `Employees` e uma para `Persons` (a fim de obedecer à interface). Depois, vem uma implementação de `StockOptionProcessingCallback` que é utilizada para manipular as opções de ação de processamento para as instâncias `Manager`.

É preciso algum tempo para entender as classes anônimas internas, mas elas são extremamente úteis. Utilizo-as o tempo todo no meu código Java. À medida que progride como desenvolvedor de Java, acho que fará a mesma coisa.

Expressões regulares

Uma *expressão regular* é, essencialmente, um padrão para descrever um conjunto de sequências de caracteres que compartilham esse padrão. Se você é um programador de Perl, deverá ficar à vontade com a sintaxe padrão de expressão regular (regex) na linguagem Java. Caso não esteja acostumado com a sintaxe de expressões regulares, porém, pode parecer estranho. Esta seção funciona como uma introdução ao uso de expressões regulares nos seus programas Java.

A API Regular Expressions

Este é um conjunto de sequências de caracteres que têm algumas coisas em comum:

- Uma cadeia de caracteres
- Uma cadeia de caracteres mais longa
- Uma cadeia de caracteres muito mais longa

Observe que cada uma dessas sequências de caracteres começa com *a* e termina com *string*. A API Java Regular Expressions ajuda a extrair todos esses elementos, ver o padrão entre eles e fazer coisas interessantes com as informações encontradas.

A API Regular Expressions possui três classes principais que você usa quase o tempo todo:

- `Pattern` descreve um padrão de cadeia de caracteres.
- `Matcher` testa uma cadeia de caracteres para ver se corresponde ao padrão.
- `PatternSyntaxException` informa que algo não era aceitável em relação ao padrão que você tentou definir.

Você começa a trabalhar em um padrão de expressões regulares simples que usa essas classes brevemente. Antes de fazer isso, porém, observe a sintaxe do padrão de expressão regular.

Sintaxe do padrão de expressão regular

Um *padrão de expressão regular* descreve a estrutura da cadeia de caracteres que a expressão tenta localizar em uma sequência de entrada. É aqui que as expressões regulares podem parecer um pouco estranhas. Contudo, depois de entender a sintaxe, fica mais fácil decifrar. A Tabela 2 lista algumas das construções de regex mais comuns que você usa em sequências padrão:

Construções de expressão regular comum

Construção de expressão regular	**O que se qualifica como uma correspondência**
`.`	Qualquer caractere
`?`	Zero (0) ou um (1) do que veio antes
`*`	Zero (0) ou mais do que veio antes
`+`	Um (1) ou mais do que veio antes
`[]`	Um intervalo de caracteres ou dígitos
`^`	Negação do que vem a seguir (ou seja, “não *qualquer coisa*“)
`\d`	Qualquer dígito (alternativamente, `[0-9]`)
`\D`	Qualquer não dígito (alternativamente, `[^0-9]`)
`\s`	Qualquer caractere de espaço em branco (alternativamente, `[\n\t\f\r]`)
`\S`	Qualquer caractere de não espaço em branco (alternativamente, `[^\n\t\f\r]`)
`\w`	Qualquer caractere de palavra (alternativamente, `[a-zA-Z_0-9]`)
`\W`	Qualquer caractere de não palavra (alternativamente, `[^\w]`)

As primeiras construções são chamadas de *quantificadores*, porque quantificam o que vem antes delas. Construções como `\d` são classes de caracteres predefinidas. Qualquer caractere que não possui um significado especial em um padrão é literal e corresponde a si mesmo.

Reconhecimento de padrões

Dispondo da sintaxe padrão da Construções de expressão regular comum, é possível lidar com o exemplo simples da Listagem 21, usando as classes na API Java Regular Expressions.

Reconhecimento de padrões com expressão regular

```
Pattern pattern = Pattern.compile("a.*string");
  Matcher matcher = pattern.matcher("a string");
  boolean didMatch = matcher.matches();
  Logger.getAnonymousLogger().info (didMatch);
  int patternStartIndex = matcher.start();
  Logger.getAnonymousLogger().info (patternStartIndex);
```

```
int patternEndIndex = matcher.end();
Logger.getAnonymousLogger().info (patternEndIndex);
```

Mostrar mais

Em primeiro lugar, a Reconhecimento de padrões com expressão regular cria uma classe `Pattern` ao chamar `compile()`, que é um método estático em `Pattern`, com uma sequência literal representando o padrão que você deseja corresponder. Essa literal utiliza a sintaxe de padrão de expressão regular. Neste exemplo, a tradução do padrão é:

_Localize uma cadeia de caracteres do formulário a seguida por zero ou mais caracteres, seguida por `string`.

Métodos para a correspondência

Em seguida, a Reconhecimento de padrões com expressão regular chama `matcher()` em `Pattern`. Essa chamada cria uma instância `Matcher`. O `Matcher` procura a cadeia de caracteres que você passou em busca de correspondências com relação à sequência padrão utilizada ao criar `Pattern`.

Cada cadeia de caracteres da linguagem Java é uma coleção indexada de caracteres, começando com 0 e terminando com o comprimento da cadeia de caracteres menos um. O `Matcher` analisa a cadeia de caracteres, começando em 0, e procura por correspondências com relação a ela. Depois que esse processo é concluído, o `Matcher` contém informações sobre as correspondências encontradas (ou não encontradas) na sequência de entrada. É possível acessar essas informações chamando vários métodos em `Matcher`:

- `matches()` informa se a sequência de entrada inteira foi uma correspondência exata para o padrão.
- `start()` informa o valor de índice na cadeia de caracteres em que a cadeia de caracteres correspondente começa.
- `end()` informa o valor de índice na cadeia de caracteres em que a cadeia de caracteres correspondente termina, mais um.

Reconhecimento de padrões com expressão regular localiza uma única correspondência começando em 0 e terminando em 7. Consequentemente, a chamada para `matches()` gera `true`, a chamada para `start()` gera `0` e a chamada para `end()` gera `8`.

`lookingAt()` versus `matches()`

Se houvesse mais elementos na sua cadeia de caracteres do que os caracteres no padrão procurado, você poderia usar `lookingAt()` em vez de `matches()`. `lookingAt()` procura correspondências de subcadeia de caracteres para um padrão específico. Considere, por exemplo, a cadeia de caracteres a seguir:

```
Esta é uma cadeia de caracteres que
não contém apenas o padrão.
```

Mostrar mais

Nela, você poderia procurar `a.*string` e obter uma correspondência se utilizar `lookingAt()`. Porém, se utilizar `matches()`, seria gerado `false`, porque há mais na cadeia de caracteres do que no padrão.

Padrões complexos na expressão regular

Procuras simples são fáceis com as classes de expressão regular, mas é possível também fazer algumas coisas altamente sofisticadas com a API Regular Expressions.

Wikis (sistemas baseados na web que permitem que os usuários modifiquem páginas) se baseiam quase que inteiramente em expressões regulares. O conteúdo da wiki é baseado na entrada de cadeia de caracteres dos usuários, que é analisada e formatada usando expressões regulares. Qualquer usuário pode criar um link para outro tópico em uma wiki ao inserir uma palavra de wiki que, normalmente, é uma série de palavras concatenadas, sendo que cada uma delas começa com uma letra maiúscula, como:

```
MyWikiWord
```

Mostrar mais

Suponha que um usuário insere a cadeia de caracteres a seguir:

```
Aqui há uma PalavraDeWiki
seguida por OutraPalavraDeWiki e, em seguida, MaisUmaPalavraDeWiki.
```

Mostrar mais

É possível procurar palavras de wiki nesta cadeia de caracteres com um padrão de expressão regular como este:

```
[A-Z][a-z]*([A-Z][a-z]*)+
```

Mostrar mais

Aqui há um código para procurar palavras de wiki:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
while (matcher.find()) {
  Logger.getAnonymousLogger().info("Found this wiki word: " + matcher.group());
}
```

Mostrar mais

Se executar esse código, você deverá ver as três palavras de wiki no seu console.

Substituindo sequências de caracteres

Procurar correspondências é útil, mas é possível também manipular sequências de caracteres após encontrar uma correspondência para elas. Para fazer isso, substitua sequências de caracteres correspondentes por outra coisa, assim como poderia procurar algum texto em um programa de processamento de texto e substituí-lo por outro texto. `Matcher` possui alguns métodos para substituir elementos de cadeia de caracteres:

- `replaceAll()` substitui todas as correspondências com uma cadeia de caracteres especificada.
- `replaceFirst()` substitui apenas a primeira correspondência por uma cadeia de caracteres especificada.

A utilização do método do `Matcher` chamado `replace` é simples:

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
Logger.getAnonymousLogger().info("Before: " + input);
String result = matcher.replaceAll("replacement");
Logger.getAnonymousLogger().info("After: " + result);
```

Mostrar mais

Esse código localiza palavras de wiki, como antes. Quando o `Matcher` localiza uma correspondência, substitui o texto da palavra de wiki pelo seu substituto. Quando executa o código, você deve ver o seguinte no seu console:

```
Antes:
Aqui há uma PalavraDeWiki seguida por OutraPalavraDeWiki e, em
seguida, AlgumaPalavraDeWiki.
  Depois: Aqui há um substituto seguido por um substituto e, em
seguida, um substituto.
```

Mostrar mais

Caso tivesse usado `replaceFirst()`, veria isto:

```
Antes: Aqui há uma PalavraDeWiki seguida por
OutraPalavraDeWiki e, em seguida, AlgumaPalavraDeWiki.
  Depois: Aqui há um substituto seguido por OutraPalavraDeWiki e,
em seguida, AlgumaPalavraDeWiki.
```

Mostrar mais

Correspondendo e manipulando grupos

Quando você procura correspondências com relação a um padrão de expressão regular, é possível obter informações sobre o que encontrou. Um pouco dessa capacidade foi visto com os métodos `start()` e `end()` no `Matcher`. Contudo, é possível também fazer referência a correspondências capturando *grupos*.

Em cada padrão, você normalmente cria grupos ao delimitar suas partes entre parênteses. Os grupos são numerados da esquerda para a direita, começando com 1 (o grupo 0 representa a correspondência inteira). O código na Listagem 22 substitui cada palavra de wiki por uma cadeia de caracteres que “agrupa” a palavra:

Correspondendo grupos

```
String input = "Here is a WikiWord followed by AnotherWikiWord, then SomeWikiWord.";
Pattern pattern = Pattern.compile("[A-Z][a-z]*([A-Z][a-z]*)+");
Matcher matcher = pattern.matcher(input);
Logger.getAnonymousLogger().info("Before: " + input);
String result = matcher.replaceAll("blah$0blah");
Logger.getAnonymousLogger().info("After: " + result);
```

Mostrar mais

Execute o código da Correspondendo grupos e obtenha a saída de console a seguir:

```
Antes: Aqui há uma PalavraDeWiki seguida por OutraPalavraDeWiki e, em
seguida, AlgumaPalavraDeWiki.
  Depois: Aqui há uma bláPalavradeWikiblá seguida por
bláOutraPalavraDeWikiblá
e, em  seguida, bláAlgumaPalavraDeWikiblá.
```

Mostrar mais

Outra abordagem para a correspondência de grupos

Correspondendo grupos faz referência à correspondência inteira ao incluir `$0` na cadeia de caracteres de substituição. Qualquer parte de uma cadeia de caracteres de substituição do formulário `$ _int` se refere ao grupo identificado pelo número inteiro (portanto, `$1` se refere ao grupo 1, etc.). Em outras palavras, `$0` é equivalente a `matcher.group(0);`.

É possível atingir o mesmo objetivo de substituição utilizando alguns outros métodos. Em vez de chamar `replaceAll()`, você poderia fazer isto:

```
StringBuffer buffer = new StringBuffer();
while (matcher.find()) {
  matcher.appendReplacement(buffer, "blah$0blah");
}
matcher.appendTail(buffer);
Logger.getAnonymousLogger().info("After: " + buffer.toString());
```

Mostrar mais

E obterá o mesmo resultado:

```
Antes: Aqui há uma PalavraDeWiki seguida por OutraPalavraDeWiki e, em
seguida, AlgumaPalavraDeWiki.
  Depois: Aqui há uma bláPalavradeWikiblá seguida por
bláOutraPalavraDeWikiblá e, em  seguida, bláAlgumaPalavraDeWikiblá.
```

Mostrar mais

Genéricos

A introdução de genéricos no JDK 5 marcou um enorme avanço para a linguagem Java. Se você usou modelos C++, perceberá que esses genéricos na linguagem Java são semelhantes, mas não exatamente iguais. Caso não tenha usado modelos C++, não se preocupe: Esta seção oferece uma introdução de alto nível a genéricos na linguagem Java.

O que são genéricos?

Com a liberação do JDK 5, a linguagem Java produziu de repente uma sintaxe nova que é estranha e interessante. Basicamente, algumas classes de JDK familiares foram substituídas pelos seus equivalentes genéricos.

Genéricos são um mecanismo de compilador em que é possível criar (e usar) tipos de coisas (como classes ou interfaces) de maneira genérica ao coletar o código comum e *parametrizar* (ou *modelar*) o resto.

Genéricos em ação

Para ver a diferença que os genéricos fazem, pense no exemplo de uma classe que está no JDK há muito tempo: `java.util.ArrayList`, que é uma `List` de `Objects` apoiada em uma array.

A Listagem 23 mostra como `java.util.ArrayList` é instanciado.

Instanciando `ArrayList`

```
ArrayList arrayList = new ArrayList();
arrayList.add("A String");
arrayList.add(new Integer(10));
arrayList.add("Another String");
// So far, so good.
```

Mostrar mais

Como é possível ver, `ArrayList` é heterogêneo: Contém dois tipos de `String` e um tipo de `Integer`. Antes do JDK 5, a linguagem Java não tinha nada para restringir esse comportamento, o que causou muitos erros de codificação. Na Instanciando `ArrayList`, por exemplo, tudo parece estar bem até o momento. Mas, em relação a acessar os elementos de `ArrayList`, o que a Listagem 24 tenta fazer?

Uma tentativa de acessar elementos em `ArrayList`

```
ArrayList arrayList = new ArrayList();
arrayList.add("A String");
arrayList.add(new Integer(10));
arrayList.add("Another String");
// So far, so good.
*processArrayList(arrayList);
*// In some later part of the code...
private void processArrayList(ArrayList theList) {
  for (int aa = 0; aa < theList.size(); aa++) {
    // At some point, this will fail...
    String s = (String)theList.get(aa);
  }
}
```

Mostrar mais

Sem conhecimento prévio do que está em `ArrayList`, você deve verificar o elemento que deseja acessar para ver se é possível manipular o tipo ou lidar com uma possível `ClassCastException`.

Com os genéricos, é possível especificar o tipo de item que entrou em `ArrayList`. A Listagem 25 mostra como.

Uma segunda tentativa, utilizando genéricos

```
ArrayList<String> arrayList = new ArrayList<>();
arrayList.add("A String");
arrayList.add(new Integer(10));// compiler error!
arrayList.add("Another String");
// So far, so good.
*processArrayList(arrayList);
*// In some later part of the code...
private void processArrayList(ArrayList<String> theList) {
  for (int aa = 0; aa < theList.size(); aa++) {
    // No cast necessary...
    String s = theList.get(aa);
  }
}
```

Mostrar mais

Iterando com genéricos

Os genéricos aprimoram a linguagem Java com uma sintaxe especial para lidar com entidades, tais como `List` s, que você normalmente deseja percorrer elemento por elemento. Se você quiser iterar por meio de `ArrayList`, por exemplo, poderia reescrever o código da Uma segunda tentativa, utilizando genéricos como:

```
private void processArrayList(ArrayList<String> theList) {
  for (String s : theList) {
    String s = theList.get(aa);
  }
}
```

Mostrar mais

Essa sintaxe funciona para qualquer tipo de objeto que é do tipo `Iterable` (ou seja, implementa a interface `Iterable`).

Classes parametrizadas

As classes parametrizadas se destacam no que se refere às coleções; portanto, observe-as nesse contexto. Considere a interface `List` (real), que representa uma coleção ordenada de objetos. No caso de uso mais comum, você inclui itens em `List` e, a seguir, acessa esses itens por índice ou por iteração em `List`.

Se estiver pensando em parametrizar uma classe, considere se os critérios a seguir se aplicam:

- Uma classe principal fica no centro de algum tipo de wrapper. Ou seja, a “coisa” no centro da classe poderia ser aplicada amplamente e os recursos (atributos, por exemplo) que a cercam são idênticos.
- Comportamento comum: Você faz praticamente as mesmas operações, independentemente da “coisa” que está no centro da classe.

Aplicando esses dois critérios, é óbvio que uma coleção se encaixa:

- A “coisa” é a classe da qual a coleção é composta.
- As operações (tais como `add`, `remove`, `size` e `clear`) são praticamente iguais, independentemente do objeto do qual a coleção é composta.

Parametrização de `List`

Na sintaxe de genéricos, o código para criar uma `List` se parece com este:

```
List<E> listReference = new concreteListClass<E>();
```

Mostrar mais

O `E`, que significa Elemento, é a “coisa” que mencionei antes. O `concreteListClass` é a classe do JDK que você está instanciando. O JDK inclui várias implementações de `List<E>`, mas você utiliza `ArrayList<E>`. Outra forma de ver uma classe genérica discutida é `Class<T>`, em que `T` significa Tipo. Quando você vê `E` em código Java, geralmente está se referindo a algum tipo de coleção. Quando vê `T`, está denotando uma classe parametrizada.

Portanto, para criar uma `ArrayList` de, por exemplo, `java.lang.Integer`, faça isto:

```
List<Integer> listOfIntegers = new ArrayList<Integer>();
```

Mostrar mais

`SimpleList`: Uma classe parametrizada

Agora, suponha que deseja criar sua própria classe parametrizada chamada `SimpleList`, com três métodos:

- `add()` inclui um elemento no fim de `SimpleList`.
- `size()` gera o número atual de elementos em `SimpleList`.
- `clear()` limpa completamente os conteúdos de `SimpleList`.

A Listagem 26 mostra a sintaxe para parametrizar `SimpleList`:

Parametrizando `SimpleList`

```
package com.makotojava.intro;
import java.util.ArrayList;
import java.util.List;
public class SimpleList<E> {
  private List<E> backingStore;
  public SimpleList() {
    backingStore = new ArrayList<E>();
  }
  public E add(E e) {
    if (backingStore.add(e))
    return e;
    else
    return null;
  }
  public int size() {
```

Mostrar mais

`SimpleList` pode ser parametrizada com qualquer subclasse `Object`. Para criar e usar uma `SimpleList` de, por exemplo, objetos `java.math.BigDecimal`, você poderia fazer isto:

```
package com.makotojava.intro;
import java.math.BigDecimal;
import java.util.logging.Logger;
import org.junit.Test;
public class SimpleListTest {
  @Test
  public void testAdd() {
    Logger log = Logger.getLogger(SimpleListTest.class.getName());

    SimpleList<BigDecimal> sl = new SimpleList<>();
    sl.add(BigDecimal.ONE);
    log.info("SimpleList size is : " + sl.size());
    sl.add(BigDecimal.ZERO);
    log.info("SimpleList size is : " + sl.size());
    sl.clear();
```

Mostrar mais

E receberia esta saída:

```
Sep 20, 2015 10:24:33 AM com.makotojava.intro.SimpleListTest testAdd
INFO: SimpleList size is: 1 Sep 20, 2015 10:24:33 AM com.makotojava.intro.SimpleListTest testA
INFO: SimpleList size is: 2 Sep 20,
2015 10:24:33 AM com.makotojava.intro.SimpleListTest testAdd
INFO: SimpleList size is: 0
```

Mostrar mais

`enum` e seus tipos

No JDK 5, foi incluído um novo tipo de dados na linguagem Java, chamado `enum`. Não deve ser confundido com `java.util.Enumeration`, `enum` representa um conjunto de objetos constantes que estão relacionados a um conceito específico; cada um representa um valor constante diferente nesse conjunto. Antes de `enum` ser introduzido na linguagem Java, você teria definido um conjunto de valores constantes para um conceito (por exemplo, gender), desta forma:

```
public class Person {
  public static final String MALE = "male";
  public static final String FEMALE = "female";
}
```

Mostrar mais

O código necessário para fazer referência a esse valor constante seria escrito mais ou menos assim:

```
public void myMethod() {
  //. . .
  String genderMale = Person.MALE;
  //. . .
}
```

Mostrar mais

Definindo constantes com `enum`

A utilização do tipo `enum` torna a definição de constantes algo muito mais formal, além de mais eficiente. Esta é a definição de `enum` para `Gender`:

```
public enum Gender {
  MALE,
  FEMALE
}
```

Mostrar mais

Esse exemplo é apenas uma pequena amostra do que é possível fazer com `enum` s. Na verdade, `enums` são muito parecidos com classes; logo, podem ter construtores, atributos e métodos:

```
package com.makotojava.intro;

public enum Gender {
  MALE("male"),
  FEMALE("female");

  private String displayName;
  private Gender(String displayName) {
    this.displayName = displayName;
  }

  public String getDisplayName() {
    return this.displayName;
  }
}
```

Mostrar mais

Uma diferença entre uma classe e um `enum` é que o construtor de um `enum` precisa ser declarado como `private` e não pode ampliar (ou herdar de) outros `enum` s. Entretanto, um `enum`pode_ implementar uma interface.

Um `enum` implementa uma interface

Suponha que você definiu uma interface como `Displayable`:

```
package com.makotojava.intro;
public interface Displayable {
  public String getDisplayName();
}
```

Mostrar mais

Seu `Gender enum` poderia implementar essa interface (e qualquer outro `enum` necessário para produzir um nome de exibição simples), desta forma:

```
package com.makotojava.intro;

public enum Gender implements Displayable {
  MALE("male"),
  FEMALE("female");

  private String displayName;
  private Gender(String displayName) {
    this.displayName = displayName;
  }
  @Override
  public String getDisplayName() {
    return this.displayName;
  }
}
```

Mostrar mais

E/S

Esta seção é uma visão geral do pacote `java.io`. Você aprende a usar algumas das suas ferramentas para coletar e manipular dados de várias origens.

Trabalhando com dados externos

Com muita frequência, os dados que você usa nos seus programas Java vêm de uma origem de dados externa, como um banco de dados, transferência direta de bytes por soquete ou armazenamento de arquivos. A linguagem Java lhe oferece muitas ferramentas para obter informações dessas origens e a maioria delas está localizada no pacote `java.io`.

Arquivos

De todas as origens de dados disponíveis para seus aplicativos Java, os arquivos são a mais comum e, muitas vezes, a mais conveniente. Se deseja ler um arquivo no seu aplicativo Java, deve-se usar *fluxos* que analisam os bytes recebidos em tipos de linguagem Java.

`java.io.File` é uma classe que define um recurso no seu sistema de arquivos e representa tal recurso de maneira abstrata. Criar um objeto `File` é fácil:

```
File f = new File("temp.txt");
File f2 = new File("/home/steve/testFile.txt");
```

Mostrar mais

O construtor `File` assume o nome do arquivo que criou. A primeira chamada cria um arquivo denominado temp.txt no diretório especificado. A segunda chamada cria um arquivo em um local específico no meu sistema Linux. É possível passar qualquer um `String` para o construtor de `File`, desde que seja um nome do arquivo válido para seu sistema operacional, independentemente de o arquivo referenciado existir ou não.

Esse código pergunta ao objeto `File` criado recentemente se o arquivo existe:

```
File f2 = new File("/home/steve/testFile.txt");
if (f2.exists()) {
  // File exists. Process it...
} else {
  // File doesn't exist. Create it...
  f2.createNewFile();
}
```

Mostrar mais

`java.io.File` possui alguns outros métodos úteis que você pode usar para excluir arquivos; criar diretórios (passando um nome de diretório como argumento para o construtor de `File`); determinar se um recurso é um arquivo, diretório ou link simbólico; e muito mais.

A ação real da E/S Java é composição e leitura a partir de origens de dados; aí que entram os fluxos.

Usando fluxos na E/S de Java

É possível acessar arquivos no sistema de arquivos utilizando fluxos. No nível mais baixo, os fluxos permitem que um programa receba bytes de uma origem ou envie a saída para um destino. Alguns fluxos manipulam todos os tipos de caracteres de 16 bits (tipos `Reader` e `Writer`). Outros manipulam somente bytes de 8 bits (tipos `InputStream` e `OutputStream`). Existem vários tipos de fluxos dentro dessas hierarquias, todos localizados no pacote `java.io`. No nível mais alto de abstração há *fluxos de caracteres* e *fluxos de bytes*.

Os fluxos de bytes leem (`InputStream` e subclasses) e escrevem (`OutputStream` e subclasses) bytes de 8 bits. Em outras palavras, um fluxo de bytes pode ser considerado um tipo de fluxo mais bruto. Este é um resumo de dois fluxos de bytes comuns e seu uso:

- **FileInputStream** / **FileOutputStream**: Lê bytes de um arquivo, escreve bytes em um arquivo
- **ByteArrayInputStream** / **ByteArrayOutputStream**: Lê bytes de uma array na memória, escreve bytes em uma array na memória

Fluxos de caracteres

Os fluxos de caracteres leem (`Reader` e subclasses) e escrevem (`Writer` e subclasses) caracteres de 16 bits. Esta é uma listagem selecionada de fluxos de caracteres e seu uso:

- `**StringReader**` / `**StringWriter**`: Leia e escreva caracteres de e para `String`s na memória.
- `**InputStreamReader**` / `**InputStreamWriter**` (e subclasses `**FileReader**` / `**FileWriter**`): Forme uma ponte entre fluxos de bytes e fluxos de caracteres. Os tipos de `Reader` leem bytes de um fluxo de bytes e os convertem em caracteres. Os tipos de `Writer` convertem caracteres em bytes para inseri-los em fluxos de byte.
- **BufferedReader** / **BufferedWriter**: Os dados ficam em buffer durante a leitura ou composição de outro fluxo, tornando as operações de leitura ou composição mais eficientes.

Em vez de tentar cobrir os fluxos na íntegra, eu foco nos fluxos recomendados para arquivos de leitura e composição. Na maioria dos casos, são fluxos de caracteres.

Lendo de um `File`

É possível ler de um `File` de várias maneiras. Sem dúvida, a abordagem mais simples é:

1. Criar um `InputStreamReader` no `File` a partir do qual deseja ler.
2. Chamar `read()` para ler um caractere por vez até chegar ao fim do arquivo.

A Listagem 27 é um exemplo de leitura de um `File`:

Lendo de um `File`

```
public List<Employee> readFromDisk(String filename) {
  final String METHOD_NAME = "readFromDisk(String filename)";
  List<Employee> ret = new ArrayList<>();
  File file = new File(filename);
  try (InputStreamReader reader = new InputStreamReader(new FileInputStream(file))) {
    StringBuilder sb = new StringBuilder();
    int numberOfEmployees = 0;
    int character = reader.read();
    while (character != -1) {
        sb.append((char)character);
        character = reader.read();
    }
    log.info("Read file: \n" + sb.toString());
    int index = 0;
    while (index < sb.length()-1) {
```

Mostrar mais

Escrevendo em um `File`

Assim como a leitura a partir de um `File`, existem várias formas de escrever em um `File`. Novamente, escolho a abordagem mais simples:

1. Criar um `FileOutputStream` no `File` no qual deseja escrever.
2. Chamar `write()` para escrever a cadeia de caracteres.

A Listagem 28 é um exemplo de composição em um `File`:

Escrevendo em um `File`

```
public boolean saveToDisk(String filename, List<Employee> employees) {
  final String METHOD_NAME = "saveToDisk(String filename, List<Employee> employees)";

  boolean ret = false;
  File file = new File(filename);
  try (OutputStreamWriter writer = new OutputStreamWriter(new FileOutputStream(file))) {
    log.info("Writing " + employees.size() + " employees to disk (as String)...");
    for (Employee employee : employees) {
      writer.write(employee.getState()+"\n");
    }
    ret = true;
    log.info("Done.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
```

Mostrar mais

Buffer de fluxos

A leitura e composição de fluxos de caracteres um caractere de cada vez não é exatamente eficiente; portanto, na maioria dos casos, provavelmente é melhor usar E/S armazenada em buffer. Para ler de um arquivo usando E/S armazenada em buffer, o código é semelhante à Lendo de um `File`, exceto pelo fato de que você agrupa `InputStreamReader` em um `BufferedReader`, como mostrado na Listagem 29.

Lendo de um `File` com E/S armazenada em buffer

```
public List<Employee> readFromDiskBuffered(String filename) {
  final String METHOD_NAME = "readFromDisk(String filename)";
  List<Employee> ret = new ArrayList<>();
  File file = new File(filename);
  try (BufferedReader reader = new BufferedReader(new InputStreamReader(new FileInputStream(fi
    String line = reader.readLine();
    int numberOfEmployees = 0;
    while (line != null) {
      StringTokenizer strtok = new StringTokenizer(line, Person.STATE_DELIMITER);
      Employee employee = new Employee();
      employee.setState(strtok);
      log.info("Read Employee: " + employee.toString());
      ret.add(employee);
      numberOfEmployees++;
      // Read next line
```

Mostrar mais

A composição em um arquivo usando E/S armazenada em buffer é igual: Você agrupa o `OutputStreamWriter` em um `BufferedWriter`, como mostrado na Listagem 30.

Escrevendo em um `File` com E/S armazenada em buffer

```
public boolean saveToDiskBuffered(String filename, List<Employee> employees) {
  final String METHOD_NAME = "saveToDisk(String filename, List<Employee> employees)";

  boolean ret = false;
  File file = new File(filename);
  try (BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(new FileOutputStream(
    log.info("Writing " + employees.size() + " employees to disk (as String)...");
    for (Employee employee : employees) {
      writer.write(employee.getState()+"\n");
    }
    ret = true;
    log.info("Done.");
  } catch (FileNotFoundException e) {
    log.logp(Level.SEVERE, SOURCE_CLASS, METHOD_NAME, "Cannot find file " +
       file.getName() + ", message = " + e.getLocalizedMessage(), e);
```

Mostrar mais

Apresentei somente uma amostra do que é possível com essa biblioteca Java essencial. Por conta própria, tente aplicar o que aprendeu sobre os arquivos em outras origens de dados.

Serialização Java

A serialização Java é mais uma das bibliotecas essenciais da plataforma Java. A serialização é usada principalmente para persistência de objeto e objeto remoto – dois casos de uso em que você precisa ser capaz de fazer uma captura instantânea do estado de um objeto e reconstituir mais tarde. Esta seção oferece uma amostra da API Java Serialization e demonstra como utilizá-la nos seus programas.

O que é a serialização de objetos?

Serialização é um processo no qual o estado de um objeto e seus metadados (como o nome da classe de objeto e os nomes dos seus atributos) são armazenados em um formato binário especial. Colocar o objeto nesse formato — *serializando-o* — preserva todas as informações necessárias para reconstituir (ou *desserializar*) o objeto sempre que precisar.

Existem dois casos de uso principais para a serialização do objeto:

- *Persistência do objeto* — armazenar o estado do objeto em um mecanismo de persistência permanente, como um banco de dados
- *Objeto remoto* — enviar o objeto para outro computador ou sistema

`java.io.Serializable`

A primeira etapa para a serialização funcionar é permitir que seus objetos usem o mecanismo. Cada objeto que você quer que seja serializável precisa implementar uma interface chamada `java.io.Serializable`:

```
import java.io.Serializable;
public class Person implements Serializable {
  // etc...
}
```

Mostrar mais

A interface `Serializable` marca o objeto da classe `Person` — e todas as subclasses de `Person` — no tempo de execução como *serializável*.

Os atributos de um objeto que não são serializáveis fazem com que o Java Runtime lance uma `NotSerializableException` se tentar serializar seu objeto. É possível gerenciar esse comportamento usando a palavra-chave `transient` para instruir o tempo de execução a não tentar serializar alguns atributos. Nesse caso, você é responsável por garantir que os atributos sejam restaurados de modo que seu objeto funcione corretamente.

Serializando um objeto

Agora, teste um exemplo que combina aquilo que você acabou de aprender sobre E/S Java com aquilo que está aprendendo a respeito da serialização.

Suponha que você criou e preencheu uma `List` de objetos `Employee` e, em seguida, deseja serializar essa `List` para um `OutputStream`; neste caso, para um arquivo. Esse processo é mostrado na Listagem 31.

Serializando um objeto

```
public class HumanResourcesApplication {
  private static final Logger log = Logger.getLogger(HumanResourcesApplication.class.getName(,
  private static final String SOURCE_CLASS = HumanResourcesApplication.class.getName();

  public static List<Employee> createEmployees() {
    List<Employee> ret = new ArrayList<Employee>();
    Employee e = new Employee("Jon Smith", 45, 175, 75, "BLUE", Gender.MALE,
       "123-45-9999", "0001", BigDecimal.valueOf(100000.0));
    ret.add(e);
    //
    e = new Employee("Jon Jones", 40, 185, 85, "BROWN", Gender.MALE, "223-45-9999",
       "0002", BigDecimal.valueOf(110000.0));
    ret.add(e);
    //
    e = new Employee("Mary Smith", 35, 155, 55, "GREEN", Gender.FEMALE, "323-45-
```

Mostrar mais

A primeira etapa é criar os objetos, o que é feito em `createEmployees()` utilizando o construtor especializado de `Employee` alguns valores de atributo. Em seguida, criou um `OutputStream`, neste caso, um `FileOutputStream`, e, depois, chamou `writeObject()` nesse fluxo. `writeObject()` é um método que usa a serialização Java para serializar um objeto para o fluxo.

Neste exemplo, você está armazenando o objeto `List` (e seus objetos `Employee` contidos) em um arquivo, mas a mesma técnica é utilizada para qualquer tipo de serialização.

Para orientar o código na Serializando um objeto, você poderia usar um teste JUnit, como mostrado aqui (em HumanResourcesApplicationTest.java):

```
public class HumanResourcesApplicationTest {

  private HumanResourcesApplication classUnderTest;
  private List<Employee> testData;

  @Before
  public void setUp() {
    classUnderTest = new HumanResourcesApplication();
    testData = HumanResourcesApplication.createEmployees();
  }
  @Test
  public void testSerializeToDisk() {
    String filename = "employees-Junit-" + System.currentTimeMillis() + ".ser";
    boolean status = classUnderTest.serializeToDisk(filename, testData);
    assertTrue(status);
```

Mostrar mais

Desserializando um objeto

A finalidade de serializar um objeto é ser capaz de reconstituí-lo ou desserializá-lo. A Listagem 32 lê o arquivo que você acabou de serializar e desserializa seu conteúdo, restaurando, assim, o estado da `List` de objetos `Employee`.

Desserializando objetos

```
public class HumanResourcesApplication {

  private static final Logger log = Logger.getLogger(HumanResourcesApplication.class.getName()
  private static final String SOURCE_CLASS = HumanResourcesApplication.class.getName();

  @SuppressWarnings("unchecked")
  public List<Employee> deserializeFromDisk(String filename) {
    final String METHOD_NAME = "deserializeFromDisk(String filename)";

    List<Employee> ret = new ArrayList<>();
    File file = new File(filename);
```

```
    int numberOfEmployees = 0;
    try (ObjectInputStream inputStream = new ObjectInputStream(new FileInputStre
      List<Employee> employees = (List<Employee>)inputStream.readObject();
```

Mostrar mais

Novamente, para orientar o código na Desserializando objetos, você poderia usar um teste JUnit como este (de HumanResourcesApplicationTest.java):

```
public class HumanResourcesApplicationTest {

  private HumanResourcesApplication classUnderTest;

  private List<Employee> testData;

  @Before
  public void setUp() {
    classUnderTest = new HumanResourcesApplication();
  }

  @Test
  public void testDeserializeFromDisk() {
    String filename = "employees-Junit-" + System.currentTimeMillis() + ".ser";
    int expectedNumberOfObjects = testData.size();
```

Mostrar mais

Para a maioria dos fins de aplicativos, marcar seus objetos como `serializable` é tudo o que você precisa fazer no que diz respeito à serialização. Quando realmente precisar serializar e desserializar seus objetos explicitamente, é possível usar a técnica mostrada na Serializando um objeto e na Desserializando objetos. Porém, à medida que seus objetos de aplicativo evoluem e você inclui atributos neles e remove atributos deles, a serialização assume uma nova camada de complexidade.

`serialVersionUID`

No princípio da comunicação com middleware e objeto remoto, os desenvolvedores eram os principais responsáveis por controlar o “formato de ligação” dos seus objetos, o que causava inúmeros problemas conforme a tecnologia evoluía.

Suponha que você incluiu um atributo em um objeto, recompilou-o e redistribuiu o código para cada computador em um cluster de aplicativo. O objeto seria armazenado em um computador com uma versão do código de serialização, mas acessado por outros computadores que podem ter uma versão diferente do código. Quando esses computadores tentavam desserializar o objeto, coisas ruins aconteciam com frequência.

Os metadados de serialização Java — as informações incluídas no formato de serialização binário — são sofisticados e resolvem muitos dos problemas que atormentavam os primeiros desenvolvedores de middleware. No entanto, não conseguem resolver todos os problemas.

A serialização Java utiliza uma propriedade chamada `serialVersionUID` para ajudá-lo a lidar com diferentes versões de objetos em um cenário de serialização. Não é necessário declarar essa propriedade nos seus objetos; por padrão, a plataforma Java utiliza um algoritmo que calcula um valor para ela com base nos seus atributos de classe, nome da classe e posição no cluster galáctico local. Na maior parte do tempo, esse algoritmo funciona normalmente. Entretanto, se você incluir ou remover um atributo, esse valor gerado dinamicamente muda e o Java Runtime lança uma `InvalidClassException`.

Para evitar esse resultado, adquira o hábito de declarar explicitamente uma `serialVersionUID`:

```
import java.io.Serializable;
  public class Person implements Serializable {
  private static final long serialVersionUID = 20100515;
  // etc...
  }
```

Mostrar mais

Recomendo utilizar algum tipo de esquema para seu número da versão do `serialVersionUID` (utilizei a data atual no exemplo anterior). Além disso, você deve declarar `serialVersionUID` como `private static final` e do tipo `long`.

Talvez você esteja se perguntando quando deve mudar essa propriedade. A resposta curta é que você deve alterá-la sempre que fizer uma mudança incompatível com a classe, o que normalmente significa que um atributo foi removido. Se você tiver uma versão do objeto em um computador do qual o atributo foi removido e o objeto for removido para um computador com uma versão do objeto em que o atributo é esperado, a situação pode ficar estranha. Neste caso, a verificação `serialVersionUID` integrada da plataforma Java é muito útil.

Como regra básica, sempre que você incluir ou remover recursos (ou seja, atributos e métodos) de uma classe, altere sua `serialVersionUID`. É melhor obter uma `InvalidClassException` na outra extremidade da ligação do que um erro de aplicativo causado por uma mudança incompatível na classe.

Conclusão da Parte 2

O tutorial *Introdução à programação de Java* abordou uma parte significativa da linguagem Java, mas a linguagem é enorme. É impossível que um único tutorial englobe tudo.

À medida que continua aprendendo sobre a linguagem e a plataforma Java, você provavelmente desejará estudar mais tópicos como expressões regulares, genéricos e serialização Java. Por fim, talvez também queira explorar tópicos que não foram abordados neste tutorial introdutório, tais como simultaneidade e persistência. Consulte Recursos para encontrar alguns pontos de partida adequados para saber mais sobre conceitos de programação de Java, incluindo aqueles que são avançados demais para serem explorados neste formato introdutório.

Audio

O conteúdo aqui presente foi traduzido da página IBM Developer US. Caso haja qualquer divergência de texto e/ou versões, consulte o conteúdo original.

Categories

Desenvolvimento de software Java