O pacote java.lang

"Nossas cabeças são redondas para que os pensamentos possam mudar de direção." -- Francis Piacaba

Ao término desse capítulo, você será capaz de:

- utilizar as principais classes do pacote `java.lang` e ler a documentação padrão de projetos java;
- usar a classe `System` para obter informações do sistema;
- utilizar a classe `String` de uma maneira eficiente e conhecer seus detalhes;
- utilizar os métodos herdados de `Object` para generalizar seu conceito de objetos.

Pacote java.lang

Já usamos, por diversas vezes, as classes `String` e `System`. Vimos o sistema de pacotes do Java e nunca precisamos dar um `import` nessas classes. Isso ocorre porque elas estão dentro do pacote `java.lang`, que é **automaticamente importado** para você. É o **único pacote** com esta característica.

Vamos ver um pouco de suas principais classes.

Um pouco sobre a classe System

A classe `System` possui uma série de atributos e métodos estáticos. Já usamos o atributo `System.out`, para imprimir.

Olhando a documentação, você vai perceber que o atributo `out` é do tipo `PrintStream` do pacote `java.io`. Veremos sobre essa classe mais adiante. Já podemos perceber que poderíamos quebrar o `System.out.println` em duas linhas:

```
PrintStream saida = System.out;
saida.println("ola mundo!");
```

O `System` conta também com um método que simplesmente desliga a virtual machine, retornando um código de erro para o sistema operacional, é o `exit`.

```
    System.exit(0);
```

Veremos também um pouco mais sobre a classe `System` nos próximos capítulos e no apêndice de `Threads`. Consulte a documentação do Java e veja outros métodos úteis da `System`.

java.lang.Object

Todo método que precisamos receber algum parâmetro temos que declarar o tipo do mesmo. Por exemplo, no nosso método `saca` precisamos passar como parâmetro um valor do tipo `double`. Se tentarmos passar qualquer coisa diferente disso teremos um erro de compilação.

Agora vamos observar o seguinte método do próprio Java:

```
System.out.println("Olá mundo!");
```

Neste caso, o método `println` está recebendo uma `String` e poderíamos pensar que o tipo de parâmetro que ele recebe é `String`. Mas ao mesmo tempo podemos passar para esse método coisas completamente diferentes como `int`, `Conta`, `Funcionario`, `SeguroDeVida`, etc. Como esse método consegue receber tantos parâmetros de tipos diferentes?

Uma possibilidade seria o uso da sobrecarga, declarando um `println` para cada tipo de objeto diferente. Mas claramente não é isso que acontece já que conseguimos criar uma classe qualquer e invocar o método `println` passando essa nova classe como parâmetro e ele funcionaria!

Para entender o que está acontecendo, vamos considerar um método que recebe uma `Conta`:

```
public void imprimeDados(Conta conta) {
    System.out.println(conta.getTitular() + " - " + conta.getSaldo());
}
```

Esse método pode ser invocado passando como parâmetro qualquer tipo de conta que temos no nosso sistema: `ContaCorrente` e `ContaPoupanca` pois ambas são filhas de `Conta`. Se quiséssemos que o nosso método conseguisse receber qualquer tipo de objeto teríamos que ter uma classe que fosse mãe de todos esses objetos. É para isso que existe a classe `Object`!

Sempre quando declaramos uma classe, essa classe é **obrigada** a herdar de outra. Isto é, para toda classe que declararmos, existe uma superclasse. Porém, criamos diversas classes sem herdar de ninguém:

```
    class MinhaClasse {

    }
```

Quando o Java não encontra a palavra chave `extends`, ele considera que você está herdando da classe `Object`, que também se encontra dentro do pacote `java.lang`. Você até mesmo pode escrever essa herança, que é o mesmo:

```
public class MinhaClasse extends Object {

}
```

Todas as classes, sem exceção, herdam de `Object`, seja direta ou indiretamente, pois ela é a mãe, vó, bisavó, etc de qualquer classe.

Podemos também afirmar que qualquer objeto em Java é um `Object`, podendo ser referenciado como tal. Então, qualquer objeto possui todos os métodos declarados na classe `Object` e veremos alguns deles logo após o *casting*.

Métodos do java.lang.Object: equals e toString

toString

A habilidade de poder se referir a qualquer objeto como `Object` nos traz muitas vantagens. Podemos criar um método que recebe um `Object` como argumento, isto é, qualquer objeto! Por exemplo, o método `println` poderia ser implementado da seguinte maneira:

```
public void println(Object obj) {
    write(obj.toString()); // o método write escreve uma string no
console
}
```

Dessa forma, qualquer objeto que passarmos como parâmetro poderá ser impresso no console desde que ele possua o método `toString`. Para garantir que todos os objetos do Java possuam esse método, ele foi implementado na classe `Object`.

Por padrão, o método `toString` do `Object` retorna o nome da classe `@` um número de identidade:

```
Conta@34f5d74a
```

Mas e se quisermos imprimir algo diferente? Na nossa tela de detalhes de conta, temos uma caixa de seleção onde nossas contas estão sendo apresentadas com o valor do padrão do `toString`. Sempre que queremos modificar o comportamento de um método em relação a implementação herdada da superclasse, podemos sobrescrevê-lo na classe filha:

```
public abstract class Conta {
    protected double saldo;
    // outros atributos...

    @Override
    public String toString() {
        return "[titular=" + titular + ", numero=" + numero
            + ", agencia=" + agencia + "]";
```

```
    }
}
```

Agora podemos usar esse método assim:

```
    ContaCorrente cc = new ContaCorrente();
    System.out.println(cc.toString());
```

E o melhor, se for apenas para jogar na tela, você nem precisa chamar o `toString`! Ele já é chamado para você:

```
    ContaCorrente cc = new ContaCorrente();
    System.out.println(cc); // O toString é chamado pela classe
PrintStream
```

Gera o mesmo resultado!

Você ainda pode concatenar `Strings` em Java com o operador `+`. Se o Java encontra um objeto no meio da concatenação, ele também chama o `toString` dele.

```
    ContaCorrente cc = new ContaCorrente();
    System.out.println("Conta: " + cc);
```

equals

Até agora estamos ignorando o fato que podemos mais de uma conta de mesmo número e agência no nosso sistema. Atualmente, quando inserimos uma nova conta, o sistema verifica se a conta inserida é igual a alguma outra conta já cadastrada. Mas qual critério de igualdade é utilizado por padrão para fazer essa verificação?

Assim como no caso do `toString`, todos objetos do Java possuem um outro método chamado `equals` que é utilizado para comparar objetos daquele tipo. Por padrão, esse método apenas compara as referências dos objetos. Como toda vez que inserimos uma nova conta no sistema estamos fazendo `new` em algum tipo de conta, as referências nunca vão ser iguais, mesmo os dados (número e agência) sendo iguais.

Mas, e se fosse preciso comparar os atributos? Quais atributos ele deveria comparar? O Java por si só não faz isso, mas podemos sobrescrever o `equals` da classe `Object` para criarmos esse critério de comparação.

O `equals` recebe um `Object` como argumento e deve verificar se ele mesmo é igual ao `Object` recebido para retornar um `boolean`. Se você não reescrever esse método, o comportamento herdado é fazer um `==` com o objeto recebido como argumento.

```
    public abstract class Conta {
        protected double saldo;
        // outros atributos...

        public boolean equals(Object object) {
            // primeiro verifica se o outro object não é nulo
            if (object == null) {
                return false;
            }
```

```
            if (this.numero == object.numero &&
                this.agencia.equals(object.agencia)) {
                return true;
            }
            return false;
        }
    }
```

Casting de referências

No momento que recebemos uma referência para um `Object`, como vamos acessar os métodos e atributos desse objeto que imaginamos ser uma `Conta`? Se estamos referenciando-o como `Object`, não podemos acessá-lo como sendo `Conta`. O código acima não compila!

Poderíamos então atribuir essa referência de `Object` para `Conta` para depois acessar os atributos necessários? Tentemos:

```
Conta outraConta = object;
```

Nós temos certeza de que esse `Object` se refere a uma `Conta`, já que a nossa lista só imprime contas. Mas o compilador Java não tem garantias sobre isso! Essa linha acima não compila, pois nem todo `Object` é uma `Conta`.

Para realizar essa atribuição, para isso devemos "avisar" o compilador Java que realmente queremos fazer isso, sabendo do risco que corremos. Fazemos o **casting de referências**, parecido com de tipos primitivos:

```
Conta outraConta = (Conta) object;
```

O código passa a compilar, mas será que roda? Esse código roda sem nenhum problema, pois em tempo de execução a JVM verificará se essa referência realmente é para um objeto de tipo `Conta`, e está! Se não estivesse, uma exceção do tipo `ClassCastException` seria lançada.

Com isso, nosso método `equals` ficaria assim:

```
    public abstract class Conta {
        protected double saldo;
        // outros atributos...

        public boolean equals(Object object) {
            if (object == null) {
                return false;
            }

            Conta outraConta = (Conta) object;
            if (this.numero == outraConta.numero &&
                this.agencia.equals(outraConta.agencia)) {
                return true;
            }
            return false;
        }
    }
```

Você poderia criar um método com outro nome em vez de reescrever `equals` que recebe `Object`, mas ele é importante pois muitas bibliotecas o chamam através do polimorfismo, como veremos no capítulo do `java.util`.

O método `hashCode()` anda de mãos dadas com o método `equals()` e é de fundamental entendimento no caso de você utilizar suas classes com estruturas de dados que usam tabelas de espalhamento. Também falaremos dele no capítulo de `java.util`.

Regras para a reescrita do método equals

Pelo contrato definido pela classe `Object` devemos retornar `false` também no caso do objeto passado não ser de tipo compatível com a sua classe. Então antes de fazer o casting devemos verificar isso, e para tal usamos a palavra chave `instanceof`, ou teríamos uma exception sendo lançada.

Além disso, podemos resumir nosso `equals` de tal forma a não usar um `if`:

```
public boolean equals(Object object) {
    if (object == null) {
        return false;
    }
    if (!(object instanceof Conta)) {
        return false;
    }
    Conta outraConta = (Conta) object;
    return (this.numero == outraConta.numero &&
        this.agencia.equals(outraConta.agencia));
}
```

Exercícios: java.lang.Object

1. Como verificar se a classe `Throwable` que é a superclasse de `Exception` também reescreve o método `toString`?

   A maioria das classes do Java que são muito utilizadas terão seus métodos `equals` e `toString` reescritos convenientemente.

2. Utilize-se da documentação do Java e descubra de que classe é o objeto referenciado pelo atributo `out` da `System`.

   Repare que, com o devido `import`, poderíamos escrever:

   ```
   // falta a declaração da saída
   ________ saida = System.out;
   saida.println("ola");
   ```

   A variável `saida` precisa ser declarada de que tipo? É isso que você precisa descobrir. Se você digitar esse código no Eclipse, ele vai te sugerir um quickfix e declarará a variável para você.

   Estudaremos essa classe em um capítulo futuro.

3. Rode a aplicação e cadastre duas contas. Na tela de detalhes de conta, verifique o que aparece na caixa de seleção de conta para transferência. Por que isso acontece?
4. Reescreva o método `toString` da sua classe `Conta` fazendo com que uma mensagem mais explicativa seja devolvida. Lembre-se de aproveitar dos recursos do Eclipse para isto: digitando apenas o começo do nome do método a ser reescrito e pressionando **ctrl + espaço**, ele vai sugerir reescrever o método, poupando o trabalho de escrever a assinatura do método e cometer algum engano.

```
5. public abstract class Conta {
6.
7.   protected double saldo;
8.
9.     @Override
10.      public String toString() {
11.          return "[titular=" + titular + ", numero=" + numero
12.             + ", agencia=" + agencia + "]";
13.      }
14.    // restante da classe
15.
   }
```

   Rode a aplicação novamente, cadastre duas contas e verifique novamente a caixa de seleção da transferência. O que aconteceu?

16. Reescreva o método `equals` da classe `Conta` para que duas contas com o mesmo **número e agência** sejam consideradas iguais. Esboço:

```
17.  public abstract class Conta {
18.
19.      public boolean equals(Object obj) {
20.          if (obj == null) {
21.              return false;
22.          }
23.
24.          Conta outraConta = (Conta) obj;
25.
26.          return this.numero == outraConta.numero &&
27.              this.agencia.equals(outraConta.agencia);
28.      }
   }
```

   Você pode usar o **ctrl + espaço** do Eclipse para escrever o esqueleto do método `equals`, basta digitar dentro da classe `equ` e pressionar **ctrl + espaço**.

   Rode a aplicação e tente adicionar duas contas com o mesmo número e agência. O que acontece?

Agora é a melhor hora de aprender algo novo

Se você gosta de estudar essa apostila aberta da Caelum, certamente vai gostar dos **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum.

Conheça a Alura

java.lang.String

`String` é uma classe em Java. Variáveis do tipo `String` guardam referências a objetos, e não um valor, como acontece com os tipos primitivos.

Aliás, podemos criar uma `String` utilizando o `new`:

```
    String x = new String("fj11");
    String y = new String("fj11");
```

Criamos aqui, dois objetos diferentes. O que acontece quando comparamos essas duas referências utilizando o `==`?

```
    if (x == y) {
        System.out.println("referência para o mesmo objeto");
    }
    else {
        System.out.println("referências para objetos diferentes!");
    }
```

Temos aqui dois objetos diferentes! E, então, como faríamos para verificar se o conteúdo do objeto é o mesmo? Utilizamos o método `equals`, que foi reescrito pela `String`, para fazer a comparação de `char` em `char`.

```
    if (x.equals(y)) {
        System.out.println("consideramos iguais no critério de
igualdade");
    }
    else {
        System.out.println("consideramos diferentes no critério de
igualdade");
    }
```

Aqui, a comparação retorna verdadeiro. Por quê? Pois quem implementou a classe `String` decidiu que este seria o melhor critério de comparação. Você pode descobrir os critérios de igualdade de cada classe pela documentação.

Podemos também concatenar `Strings` usando o `+`. Podemos concatenar `Strings` com qualquer objeto, até mesmo números:

```
    int total = 5;
    System.out.println("o total gasto é: " + total);
```

O compilador utilizará os métodos apropriados da classe `String` e possivelmente métodos de outras classes para realizar tal tarefa.

Se quisermos comparar duas Strings, utilizamos o método `compareTo`, que recebe uma `String` como argumento e devolve um inteiro indicando se a `String` vem antes, é igual ou vem depois da `String` recebida. Se forem iguais, é devolvido `0`; se for anterior à

`String` do argumento, devolve um inteiro negativo; e, se for posterior, um inteiro positivo.

Fato importante: **uma String é imutável**. O java cria um pool de Strings para usar como cache e, se a `String` não fosse imutável, mudando o valor de uma `String` afetaria todas as `String`s de outras classes que tivessem o mesmo valor.

Repare no código abaixo:

```
String palavra = "fj11";
palavra.toUpperCase();
System.out.println(palavra);
```

Pode parecer estranho, mas ele imprime "fj11" em minúsculo. Todo método que parece alterar o valor de uma `String`, na verdade, cria uma nova `String` com as mudanças solicitadas e a retorna! Tanto que esse método não é `void`. O código realmente útil ficaria assim:

```
String palavra = "fj11";
String outra = palavra.toUpperCase();
System.out.println(outra);
```

Ou você pode eliminar a criação de outra variável temporária, se achar conveniente:

```
String palavra = "fj11";
palavra = palavra.toUpperCase();
System.out.println(palavra);
```

Isso funciona da mesma forma para **todos** os métodos que parecem alterar o conteúdo de uma `String`.

Se você ainda quiser trocar o número 1 para 2, faríamos:

```
String palavra = "fj11";
palavra = palavra.toUpperCase();
palavra = palavra.replace("1", "2");
System.out.println(palavra);
```

Ou ainda podemos concatenar as invocações de método, já que uma `String` é devolvida a cada invocação:

```
String palavra = "fj11";
palavra = palavra.toUpperCase().replace("1", "2");
System.out.println(palavra);
```

O funcionamento do pool interno de Strings do Java tem uma série de detalhes e você pode encontrar mais informações sobre isto na documentação da classe `String` e no seu método `intern()`.

Outros métodos da classe `String`

Existem diversos métodos da classe `String` que são extremamente importantes. Recomendamos sempre consultar o javadoc relativo a essa classe para aprender cada vez mais sobre a mesma.

Por exemplo, o método `charAt(i)`, retorna o caractere existente na posição `i` da String, o **método** `length` retorna o número de caracteres na mesma e o método `substring` que recebe um `int` e devolve a SubString a partir da posição passada por aquele `int`.

O `indexOf` recebe um char ou uma String e devolve o índice em que aparece pela primeira vez na String principal (há também o `lastIndexOf` que devolve o índice da última ocorrência).

O `toUpperCase` e o `toLowerCase` devolvem uma nova String toda em maiúscula e toda em minúscula, respectivamente.

A partir do Java 6, temos ainda o método `isEmpty`, que devolve `true` se a String for vazia ou `false` caso contrário.

Alguns métodos úteis para buscas são o `contains` e o `matches`.

Há muitos outros métodos, recomendamos que você sempre consulte o javadoc da classe.

java.lang `StringBuffer` e `StringBuilder`

Como a classe `String` é imutável, trabalhar com uma mesma `String` diversas vezes pode ter um efeito colateral: gerar inúmeras `String`s temporárias. Isto prejudica a performance da aplicação consideravelmente.

No caso de você trabalhar muito com a manipulação de uma mesma `String` (por exemplo, dentro de um laço), o ideal é utilizar a classe `StringBuffer`. A classe `StringBuffer` representa uma sequência de caracteres. Diferentemente da `String`, ela é mutável, e não possui aquele pool.

A classe `StringBuilder` tem exatamente os mesmos métodos, com a diferença dela não ser **thread-safe**. Veremos sobre este conceito no capítulo de Threads.String.

Exercícios: java.lang.String

1. Queremos que as contas apresentadas na caixa de seleção da transferência apareçam com o nome do titular em maiúsculas. Para fazer isso vamos alterar o método `toString` da classe `Conta`. Utilize o método `toUpperCase` da `String` para isso.
2. Após alterarmos o método `toString`, aconteceu alguma mudança com o nome do titular que é apresentado na lista de contas? Por que?
3. Teste os exemplos desse capítulo, para ver que uma `String` é imutável. Por exemplo:

```
4.     public class TestaString {
5.
6.         public static void main(String[] args) {
```

```
7.              String s = "fj11";
8.              s.replaceAll("1", "2");
9.              System.out.println(s);
10.         }
11.
        }
```

Como fazer para ele imprimir fj22?

12. Como fazer para saber se uma `String` se encontra dentro de outra? E para tirar os espaços em branco das pontas de uma `String`? E para saber se uma `String` está vazia? E para saber quantos caracteres tem uma `String`?

    Tome como hábito sempre pesquisar o JavaDoc! Conhecer a API, aos poucos, é fundamental para que você não precise reescrever a roda!

13. (opcional) Escreva um método que usa os métodos `charAt` e `length` de uma `String` para imprimir a mesma caractere a caractere, com cada caractere em uma linha diferente.
14. (opcional) Reescreva o método do exercício anterior, mas modificando ele para que imprima a `String` de trás para a frente e em uma linha só. Teste-a para *"Socorram-me, subi no ônibus em Marrocos"* e *"anotaram a data da maratona"*.
15. (opcional) Pesquise a classe `StringBuilder` (ou StringBuffer no Java 1.4). Ela é mutável. Por que usá-la em vez da `String`? Quando usá-la?

    Como você poderia reescrever o método de escrever a `String` de trás para a frente usando um `StringBuilder`?

Desafio

1. Converta uma `String` para um número sem usar as bibliotecas do java que já fazem isso. Isso é, uma `String x = "762"` deve gerar um `int i = 762`.

   Para ajudar, saiba que um `char` pode ser "transformado" em `int` com o mesmo valor numérico fazendo:

   ```
   char c = '3';
   int i = c - '0'; // i vale 3!
   ```

   Aqui estamos nos aproveitando do conhecimento da tabela unicode: os números de 0 a 9 estão em sequência! Você poderia usar o método estático `Character.getNumericValue(char)` em vez disso.

A Caelum oferece o **curso Java e Orientação a Objetos** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Java e Orientação a Objetos*

Discussão em aula: O que você precisa fazer em Java?

Qual é a sua necessidade com o Java? Precisa fazer algoritmos de redes neurais? Gerar gráficos 3D? Relatórios em PDF? Gerar código de barra? Gerar boletos? Validar CPF? Mexer com um arquivo do Excel?

O instrutor vai mostrar que para a maioria absoluta das suas necessidades, alguém já fez uma biblioteca e a disponibilizou.