

## CAPÍTULO 14

# 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;
- usar as classes wrappers (como `Integer`) e boxing;
- utilizar os métodos herdados de `Object` para generalizar seu conceito de objetos.

## 14.1 - 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.

## 14.2 - 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 no próximo capítulo. Já podemos perceber que poderíamos quebrar o `System.out.println` em duas linhas:

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

Ela também possui o atributo `in`, que lê da entrada padrão, porém só consegue captar bytes:

```
int i = System.in.read();
```

O código acima deve estar dentro de um bloco de `try` e `catch`, pois pode lançar uma exceção `IOException`. É útil ficar lendo de byte em byte? Trabalharemos mais com a entrada padrão também no próximo capítulo.

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` no próximo capítulo e no de `Threads`. Consulte a documentação do Java e veja outros métodos úteis da `System`.

## A classe Runtime

A classe `Runtime` possui um método para fazer uma chamada ao sistema operacional e rodar algum programa:

```
Runtime rt = Runtime.getRuntime();
Process p = rt.exec("dir");
```

Isto deve ser evitado ao máximo, já que gera uma dependência da sua aplicação com o sistema operacional em questão, perdendo a portabilidade entre plataformas. Em muitos casos, isso pode ser substituído por chamadas às bibliotecas do Java. Nesse caso, por exemplo, você tem um método `list` na classe `File` do pacote de entrada e saída, que veremos posteriormente.

O método `exec` te retorna um `Process`, onde você é capaz de pegar a saída do programa, enviar dados para a entrada, entre outros.

## Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil. Conheça os títulos e a nova proposta, você vai gostar.

[Casa do Código, livros para o programador.](#)

## 14.3 - JAVA.LANG.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:

```
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*.

## 14.4 - CASTING DE REFERÊNCIAS

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! Melhor, podemos armazenar qualquer objeto:

```
public class GuardadorDeObjetos {
```

```

private Object[] arrayDeObjetos = new Object[100];
private int posicao = 0;

public void adicionaObjeto(Object object) {
    this.arrayDeObjetos[this.posicao] = object;
    this.posicao++;
}

public Object pegaObjeto(int indice) {
    return this.arrayDeObjetos[indice];
}
}

```

Mas, e no momento que retirarmos uma referência a esse objeto, como vamos acessar os métodos e atributos desse objeto? Se estamos referenciando-o como `Object`, não podemos acessá-lo como sendo `Conta`. Veja o exemplo a seguir:

```

GuardadorDeObjetos guardador = new GuardadorDeObjetos();
Conta conta = new Conta();
guardador.adicionaObjeto(conta);

// ...

// pega a conta referenciado como objeto
Object object = guardador.pegaObjeto(0);

// será que posso invocar getSaldo em Object? :
object.getSaldo();

```

Poderíamos então atribuir essa referência de `Object` para `Conta` para depois invocar o `getSaldo()`? Tentemos:

```
Conta contaResgatada = object;
```

Nós temos certeza de que esse `Object` se refere a uma `Conta`, já que fomos nós que o adicionamos na classe que guarda objetos. 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 contaResgatada = (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.

Poderíamos fazer o mesmo com `Funcionario` e `Gerente`. Tendo uma referência para um

Funcionario que temos certeza ser um Gerente, podemos fazer a atribuição, desde que o casting exista, pois nem todo Funcionario é um Gerente.

```
Funcionario funcionario = new Gerente();
// ... e depois
Gerente gerente = funcionario; // não compila!
// nem todo Funcionario é um Gerente
```

O correto então seria:

```
Gerente gerente = (Gerente) funcionario;
```

Vamos misturar um pouco:

```
Object object = new Conta();
// ... e depois
Gerente gerente = (Gerente) object;
```

Esse código compila? Roda?

Compila, pois existe a chance de um Object ser um Gerente. Porém não roda, ele vai lançar uma Exception (ClassCastException) em tempo de execução. É importante diferenciar tempo de compilação e tempo de execução.

Neste exemplo, nós garantimos ao java que nosso Objeto object era um Gerente com o casting, por isso compilou, mas na hora de rodar, quando ele foi receber um Gerente, ele recebeu uma Conta, daí ele reclamou lançando ClassCastException!

## 14.5 - MÉTODOS DO JAVA.LANG.OBJECT: EQUALS E TOSTRING

O primeiro método interessante é o `toString`. As classes podem reescrever esse método para mostrar uma mensagem, uma `String`, que o represente. Você pode usá-lo assim:

```
Conta c = new Conta();
System.out.println(c.toString());
```

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

```
Conta@34f5d74a
```

Mas isso não é interessante para nós. Então podemos reescrevê-lo:

```
class Conta {
```

```

private double saldo;
// outros atributos...

public Conta(double saldo) {
    this.saldo = saldo;
}

public String toString() {
    return "Uma conta com valor: " + this.saldo;
}
}

```

Chamando o `toString`:

```

Conta c = new Conta(100);
System.out.println(c.toString()); // imprime: Uma conta com valor: 100.

```

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

```

Conta c = new Conta(100);
System.out.println(c); // 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.

```

Conta c = new Conta(100);
System.out.println("Descrição: " + c);

```

O outro método muito importante é o `equals`. Quando comparamos duas variáveis referência no Java, o `==` verifica se as duas referem-se ao mesmo objeto:

```

Conta c1 = new Conta(100);
Conta c2 = new Conta(100);
if (c1 != c2) {
    System.out.println("objetos referenciados são diferentes!");
}

```

E, nesse caso, realmente são diferentes.

Mas, e se fosse preciso comparar os atributos? Quais atributos ele deveria comparar? O Java por si só não faz isso, mas existe um método na classe `Object` que pode ser reescrito para criarmos esse critério de comparação. Esse método é o `equals`.

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 class Conta {
    private double saldo;
    // outros atributos...

    public Conta(double saldo) {
        this.saldo = saldo;
    }

    public boolean equals(Object object) {
        Conta outraConta = (Conta) object;
        if (this.saldo == outraConta.saldo) {
            return true;
        }
        return false;
    }

    public String toString() {
        return "Uma conta com valor: " + this.saldo;
    }
}

```

Um exemplo clássico do uso do `equals` é para datas. Se você criar duas datas, isto é, dois objetos diferentes, contendo `31/10/1979`, ao comparar com `==` receberá `false`, pois são referências para objetos diferentes. Seria correto, então, reescrever este método, fazendo as comparações dos atributos, e o usuário passaria a invocar `equals` em vez de comparar com `==`.

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`:

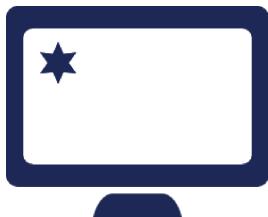
```

public boolean equals(Object object) {
    if (!(object instanceof Conta))
        return false;
    Conta outraConta = (Conta) object;
}

```

```
        return this.saldo == outraConta.saldo;
    }
```

## Agora é a melhor hora de aprender algo novo



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

[Conheça a Alura.](#)

## 14.6 - INTEGER E CLASSES WRAPPERS (BOX)

Uma pergunta bem simples que surge na cabeça de todo programador ao aprender uma nova linguagem é: "Como transformar um número em `String` e vice-versa?".

Cuidado! Usamos aqui o termo "transformar", porém o que ocorre não é uma transformação entre os tipos e sim uma forma de conseguirmos uma `String` dado um `int` e vice-versa. O jeito mais simples de transformar um número em `String` é concatená-lo da seguinte maneira:

```
int i = 100;
String s = "" + i;
System.out.println(s);
```

```
double d = 1.2;
String s2 = "" + d;
System.out.println(s2);
```

Para formatar o número de uma maneira diferente, com vírgula e número de casas decimais devemos utilizar outras classes de ajuda (`NumberFormat`, `Formatter`).

Para transformar uma `String` em número, utilizamos as classes de ajuda para os tipos primitivos correspondentes. Por exemplo, para transformar a `String` `s` em um número inteiro utilizamos o método estático da classe `Integer`:

```
String s = "101";
int i = Integer.parseInt(s);
```

As classes `Double`, `Short`, `Long`, `Float` etc contêm o mesmo tipo de método, como `parseDouble` e `parseFloat` que retornam um `double` e `float` respectivamente.

Essas classes também são muito utilizadas para fazer o **wrapping** (embrulho) de tipos primitivos como objetos, pois referências e tipos primitivos são incompatíveis. Imagine que precisamos passar como argumento um inteiro para o nosso guardador de objetos. Um inteiro não é um `Object`, como fazer?

```
int i = 5;
Integer x = new Integer(i);
guardador.adiciona(x);
```

E, dado um `Integer`, podemos pegar o `int` que está dentro dele (desembrulhá-lo):

```
int i = 5;
Integer x = new Integer(i);
int numeroDeVolta = x.intValue();
```

## 14.7 - AUTOBOXING NO JAVA 5.0

Esse processo de wrapping e unwrapping é entediante. O Java 5.0 em diante traz um recurso chamado de **autoboxing**, que faz isso sozinho para você, custando legibilidade:

```
Integer x = 5;
int y = x;
```

No Java 1.4 esse código é inválido. No Java 5.0 em diante ele compila perfeitamente. É importante ressaltar que isso não quer dizer que tipos primitivos e referências sejam do mesmo tipo, isso é simplesmente um "açúcar sintático" (*syntax sugar*) para facilitar a codificação.

Você pode fazer todos os tipos de operações matemáticas com os wrappers, porém corre o risco de tomar um `NullPointerException`.

Você pode fazer o autoboxing diretamente para `Object` também, possibilitando passar um tipo primitivo para um método que receber `Object` como argumento:

```
Object o = 5;
```

## 14.8 - 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 das classes wrappers para realizar tal tarefa.

A classe `String` conta também com um método `split`, que divide a `String` em um array de `Strings`, dado determinado critério.

```
String frase = "java é demais";
String palavras[] = frase.split(" ");
```

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 `Strings` 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.

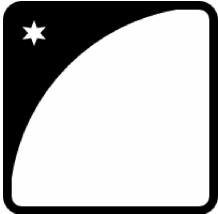
## **java.lang StringBuffer e StringBuilder**

Como a classe `String` é imutável, trabalhar com uma mesma `String` diversas vezes pode ter um efeito colateral: gerar inúmeras `Strings` 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.

**Você pode também fazer o curso FJ-11 dessa apostila na Caelum**



Querendo aprender ainda mais sobre Java e boas práticas de orientação a objetos? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso FJ-11** 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.](#)

## 14.9 - JAVA.LANG.MATH

Na classe `Math`, existe uma série de métodos estáticos que fazem operações com números como, por exemplo, arredondar(`round`), tirar o valor absoluto (`abs`), tirar a raiz(`sqr`), calcular o seno(`sin`) e outros.

```
double d = 4.6;
long i = Math.round(d);

int x = -4;
int y = Math.abs(x);
```

Consulte a documentação para ver a grande quantidade de métodos diferentes.

No Java 5.0, podemos tirar proveito do `import static` aqui:

```
import static java.lang.Math.*;
```

Isso elimina a necessidade de usar o nome da classe, sob o custo de legibilidade:

```
double d = 4.6;
long i = round(d);
int x = -4;
int y = abs(x);
```

## 14.10 - EXERCÍCIOS: JAVA.LANG

Aqui faremos diversos testes, além de modificar a classe `Conta`. Você pode fazer todos esses exercícios dentro do próprio projeto `banco`.

1. Teste os exemplos desse capítulo, para ver que uma `String` é imutável. Por exemplo:

```
public class TestaString {

    public static void main(String[] args) {
        String s = "fj11";
```

```
s.replaceAll("1", "2");
System.out.println(s);
}

}
```

Como fazer para ele imprimir f22?

2. 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!

3. Crie uma classe `TestaInteger` e vamos fazer comparações com `Integers` dentro do `main`:

```
Integer x1 = new Integer(10);
Integer x2 = new Integer(10);

if (x1 == x2) {
    System.out.println("igual");
} else {
    System.out.println("diferente");
}
```

E se testarmos com o `equals`? O que podemos concluir?

4. Como verificar se a classe `Integer` também reescreve o método `toString`?

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

Aproveite e faça um teste com o método estático `parseInt`, recebendo uma `String` válida e uma inválida (com caracteres alfabéticos), e veja o que acontece!

5. 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 no capítulo seguinte.

6. Crie e imprima uma referência de Conta. Note que você vai ter que dar new em ContaCorrente ou ContaPoupanca, já que sua Conta é abstrata:

```
Conta conta = new ContaCorrente();
System.out.println(conta);
```

O que acontece?

7. 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.

```
public abstract class Conta {
    private double saldo;
    public String toString() {
        return "esse objeto é uma conta com saldo R$" + this.saldo;
    }
    // restante da classe
}
```

Imprima novamente uma referência a Conta. O que aconteceu?

8. Reescreva o método `equals` da classe Conta para que duas contas com o mesmo **número de conta** sejam consideradas iguais. Para isso, você vai precisar de um atributo numero. Esboço:

```
public abstract class Conta {
    private int numero;
    public boolean equals(Object obj) {
        Conta outraConta = (Conta) obj;
        return this.numero == outraConta.numero;
    }
    // coloque getter e setter para numero, usando Eclipse!
}
```

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**.

Crie uma classe TestaComparacaoConta e, dentro do main, crie duas instâncias de

ContaCorrente com números iguais. Áí compare elas com `==` e depois com `equals`.

9. Um `double` não está sendo suficiente para guardar a quantidade de casas necessárias em uma aplicação. Preciso guardar um número decimal muito grande! O que poderia usar?

O `double` também tem problemas de precisão ao fazer contas, por causa de arredondamentos da aritmética de ponto flutuante definido pela IEEE 754:

[http://en.wikipedia.org/wiki/IEEE\\_754](http://en.wikipedia.org/wiki/IEEE_754)

Ele não deve ser usado se você precisa realmente de muita precisão (casos que envolvam dinheiro, por exemplo).

**Consulte a documentação**, tente adivinhar onde você pode encontrar um tipo que te ajudaria para resolver esses casos e veja como é intuitivo! Qual é a classe que resolveria esses problemas?

Lembre-se: no Java há muito já pronto. Seja na biblioteca padrão, seja em bibliotecas *open source* que você pode encontrar pela internet.

10. (opcional) Faça com que o `equals` da sua classe Conta também leve em consideração a `String` do nome do cliente a qual ela pertence. Se sua Conta não possuir o atributo `nome`, crie-o. Teste se o método criado está funcionando corretamente.

11. (opcional) Crie a classe `GuardadorDeObjetos` como visto nesse capítulo. Crie uma classe `TestaGuardador` e dentro do `main` crie uma `ContaCorrente` e adicione-a em um `GuardadorDeObjetos`. Depois teste pegar essa referência como `ContaPoupanca`, usando casting:

```
GuardadorDeObjetos guardador = new GuardadorDeObjetos();
ContaCorrente cc = new ContaCorrente();
guardador.adicionaObjeto(cc);
```

```
// vai precisar do casting para compilar!
// use Ctrl+1 para o Eclipse gerar para você
ContaPoupanca cp = guardador.pega(0);
```

Repare na exception que é lançada. Qual é o tipo dela?

Teste também o autoboxing do Java 5.0, passando um inteiro para nosso guardador.

12. (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.

13. (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".*

14. (opcional) Dada uma frase, reescreva essa frase com as palavras na ordem invertida.

"*Socorram-me, subi no ônibus em Marrocos*" deve retornar "*Marrocos em ônibus no subi Socorram-me,*". Utilize o método `split` da `String` para te auxiliar.

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`?

## 14.11 - 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.

### Tire suas dúvidas no novo GUJ Respostas



O GUJ é um dos principais fóruns brasileiros de computação e o maior em português sobre Java. A nova versão do GUJ é baseada em uma ferramenta de *perguntas e respostas* (QA) e tem uma comunidade muito forte. São mais de 150 mil usuários pra ajudar você a esclarecer suas dúvidas.

[Faça sua pergunta.](#)

## 14.12 - 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.

CAPÍTULO ANTERIOR:

[Ferramentas: jar e javadoc](#)

PRÓXIMO CAPÍTULO:

[Pacote java.io](#)

Você encontra a Caelum também em:

Blog Caelum

Cursos Online

Facebook

Newsletter

# Casa do Código

Twitter