Índice

Data desta edição: 18/10/2005

capítulo 1

Como aprender Java

"Homens sábios fazem provérbios, tolos os repetem"
Samuel Palmer -

Como o material está organizado e dicas de como estudar em casa.

1.1 - Falando em Java

Esta é a apostila da Caelum que tem como intuito ensinar Java de uma maneira elegante, mostrando apenas o que é necessário no momento correto e poupando o leitor de assuntos que não costumam ser de seu interesse em determinadas fases do aprendizado.

1.2 - O que é realmente importante?

Muitos livros, ao passar os capítulos, mencionam todos os detalhes da linguagem juntamente com os princípios básicos dela. Isso acaba criando muita confusão, em especial pois o estudante não consegue distinguir exatamente o que é importante aprender e reter naquele momento daquilo que será necessário mais tempo e principalmente experiência para dominar.

Se uma classe abstrata deve ou não ter ao menos um método abstrato, se o if só aceitar argumentos booleanos e todos os detalhes de classes internas realmente não devem ser preocupações para aquele que possui como objetivo primário aprender Java. Esse tipo de informação será adquirida com o tempo, e não é necessário até um segundo momento.

Neste curso separamos essas informações em quadros especiais, já que são informações extras. Ou então apenas citamos num exercício e deixamos para o leitor procurar informações se for de seu interesse.

Algumas informações não são mostradas e podem ser adquiridas em tutoriais ou guias de referência, normalmente são detalhes que para um programador experiente em Java é algo importante.

Por fim falta mencionar sobre a prática, que deve ser tratada seriamente: todos os exercícios são muito importantes e os desafios podem ser feitos quando o curso acabar. De qualquer maneira recomendamos aos alunos estudar em casa, principalmente aqueles que fazem os cursos intensivos.

O curso

Para aqueles que estão fazendo o curso Java e Orientação a Objetos, é recomendado estudar em casa aquilo que foi visto durante a aula, tentando resolver os exercícios que não foram feitos e os desafios que estão lá para envolver mais o leitor no mundo de Java.

Convenções de Código

Para mais informações sobre as convenções de código-fonte Java, acesse: http://java.sun.com/docs/codeconv/

1.3 - Sobre os exercícios

Os exercícios do curso variam entre práticos até pesquisas na Internet, ou mesmo consultas sobre assuntos avançados em determinados tópicos para incitar a curiosidade do aprendiz na tecnologia.

Existem também, em determinados capítulos, uma série de desafios. Eles focam mais no problema computacional que na linguagem, porém são uma excelente forma de treinar a sintaxe e principalmente familiarizar o aluno com a biblioteca padrão Java, além de o aluno ganhar velocidade de raciocínio.

1.4 - Tirando dúvidas

Para tirar dúvidas dos exercícios, ou de Java em geral, recomendamos o fórum do site do GUJ (http://www.guj.com.br/), onde sua dúvida será respondida prontamente.

Se você já participa de um grupo de usuários java ou alguma lista de discussão, pode tirar suas dúvidas nos dois lugares.

Fora isso, sinta-se a vontade de entrar em contato conosco para tirar todas as suas dúvidas durante o curso.

1.5 - Sobre o curso

A Caelum (http://www.caelum.com.br) oferece os cursos e a apostila "Falando em Java", que aborda o ensino dessa linguagem e tecnologia de forma mais simples e prática do que em outros cursos, poupando o aluno de assuntos que não são de seu interesse em determinadas fases do seu aprendizado.

As apostilas "Falando em Java" estão parcialmente disponíveis no site http://www.caelum.com.br/fj.jsp.

Se você possui alguma colaboração, como correção de erros, sugestões, novos exercícios e outros, entre em contato conosco!

1.6 - Sobre os autores

Guilherme Silveira (guilherme.silveira@caelum.com.br) é programador e web developer certificado pela Sun, trabalhando com Java desde 2000 como especialista e instrutor. Programou e arquiteturou projetos na Alemanha. Cofundador do GUJ, escreve para a revista Mundo Java, estuda Matemática Aplicada na USP e é instrutor na Caelum.

Paulo Silveira (paulo.silveira@caelum.com.br) é programador e desenvolvedor certificado Java. Possui grande experiência com servlets, que utilizou

na Alemanha, e vários outros projetos Java, onde trabalhou como consultor sênior. Foi instrutor Java pela Sun, cofundador do GUJ e criador do framework vRaptor. É formado em ciência da computação pela USP, onde realiza seu mestrado.

Sérgio Lopes (sergio@caelum.com.br) é programador e desenvolvedor Java desde 2002. É moderador do Grupo de Usuários Java – GUJ - e estuda Ciência da Computação na USP.

capítulo 2

O que é Java

"Computadores são inúteis, eles apenas dão respostas"
- Picasso

Chegou a hora de responder as perguntas mais básicas sobre Java. Ao término desse capítulo você será capaz de:

- responder o que é Java;
- mostrar as vantagens e desvantagens de Java;
- compilar e executar um programa simples.

2.1 - Java

Muitos associam Java com uma maneira de deixar suas páginas da web mais bonitas, com efeitos especiais, ou para fazer pequenos formulários na web.

O que associa as **empresas** ao **Java**?

Já iremos chegar neste ponto, mas antes vamos ver o motivo pelo qual as empresas fogem das outras linguagens:

Quais são os seus maiores problemas quando está programando?

- ponteiros?
- liberar memória?
- organização?
- falta de bibliotecas boas?
- ter de reescrever parte do código ao mudar de sistema operacional?
- custo de usar a tecnologia?

PLATAFORMA JAVA

Java tenta amenizar esses problemas. Alguns desses objetivos foram atingidos muito tempo atrás, porque, antes do java 1.0 sair, a idéia é que a linguagem fosse usada em pequenos dispositivos, como tvs, aspiradores, liquidificadores e outros. Apesar disso a linguagem teve seu lançamento mirando o uso dela nos clientes web (browsers) para rodar pequenas aplicações (applets). Hoje em dia esse não é mais o foco da linguagem.

SUN

O Java é desenvolvido e mantido pela Sun (http://www.sun.com) e seu site principal é o http://java.sun.com. (java.com é um site mais institucional, voltado ao consumidor de produtos e usuários leigos, não desenvolvedores).

A história do Java

Você pode ler a história da linguagem Java em:
http://java.sun.com/java2/whatis/1996/storyofjava.html

No Brasil, diversos grupos de usuários se juntaram para tentar disseminar o conhecimento da linguagem. Um deles é o GUJ (www.guj.com.br), uma comunidade

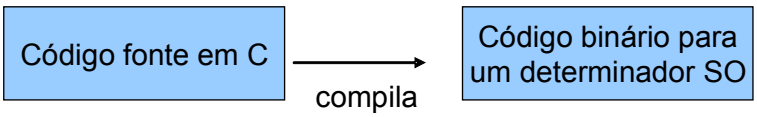
virtual com artigos, tutoriais e fórum para tirar dúvidas, o maior em língua portuguesa.

Encorajamos todos os alunos a usar muito os fóruns do mesmo pois é uma das melhores maneiras para achar soluções para pequenos problemas que acontecem com grande freqüência.

2.2 - Máquina Virtual

Em uma linguagem de programação como C e Pascal, temos o seguinte quadro quando vamos compilar um programa.

O código fonte é compilado para uma plataforma e sistema operacional específicos. Muitas vezes, o próprio código fonte é desenvolvido visando uma única plataforma!



Esse código executável (binário) resultante será executado pelo sistema operacional e, por esse motivo, ele deve saber conversar com o sistema operacional em questão.

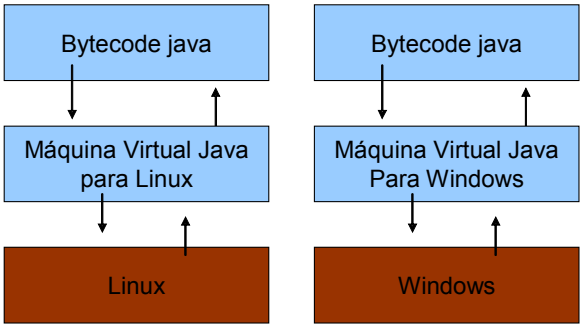
Isto é, temos um código executável para cada sistema operacional. É necessário compilar uma vez para Windows, outra para o Linux, etc...

Como foi dito anteriormente, na maioria das vezes, a sua aplicação se utiliza das bibliotecas do sistema operacional, como, por exemplo, a de interface gráfica para desenhar as 'telinhas'. A biblioteca de interface gráfica do Windows é bem diferente das do Linux; resultado?

Você precisa reescrever o mesmo pedaço da aplicação para diferentes sistemas operacionais, já que eles não são compatíveis.

MÁQUINA VIRTUAL

Já o Java utiliza-se do conceito de **máquina virtual**, onde existe uma camada extra entre o sistema operacional e a aplicação, responsável por "traduzir" (não é apenas isso) o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional no qual ela está rodando no momento:



Dessa forma, a maneira a qual você abre uma janela no Linux ou no Windows é a mesma: você ganha independência de sistema operacional. Ou, melhor ainda, independência de plataforma em geral: não é preciso se preocupar em qual sistema operacional sua aplicação está rodando, nem em que tipo de máquina, configurações etc.

BYTECODE

Mas para isso, precisamos de um "**bytecode**". Bytecode é o termo dado ao código binário gerado pelo compilador Java (pois existem menos de 255 códigos de operação dessa linguagem, e cada "opcode" gasta um byte, dando origem ao nome bytecode). O compilador Java gera esse bytecode que, diferente das linguagens sem máquina virtual, vai servir para diferentes sistemas operacionais, já que ele vai ser "traduzido" pela máquina virtual.

Write once, run anywhere

Esse é um slogan que a Sun usa para o Java, já que você não precisa reescrever parte da sua aplicação toda vez que quiser mudar de sistema operacional.

Muitas pessoas criticam ou criam piadas em cima desse slogan, por acreditarem que nem sempre uma aplicação Java pode rodar em duas plataformas diferentes sem problemas.

2.3 - Instalando o Java

Antes de instalar, baixe o **J2SDK 5.0** ou superior, do site do Java da Sun, em http://java.sun.com . Pegue a versão internacional e cuidado para não baixar o que tem mais de 90 megas, que é a primeira opção na página de download: esta versão vem com o Netbeans, que é uma ferramenta da Sun, e não nos interessa no momento. Mais para baixo da página existe uma versão menor, algo em torno de 45 megas, sem essa ferramenta.

Esse software disponível na Sun é gratuito, assim como as principais bibliotecas Java e ferramentas.

É interessante você também baixar a **documentação** do J2SDK 5.0, o link se encontra na mesma página e possui outros 40 megas.

O procedimento de instalação no Windows é muito simples: basta você executar o arquivo e seguir os passos. Instale-o no diretório desejado.

Depois disso, é necessário configurar algumas variáveis de ambiente, para que você possa executar o compilador Java e a máquina virtual de qualquer diretório. Em cada Windows você configura as variáveis de ambiente de uma maneira diferente. São duas as variáveis que você deve mudar:

```
CLASSPATH=.
PATH=<o que ja estava antes>;c:\diretorioDeInstalacaoDoJava\bin
```

CLASSPATH

A variável **PATH** provavelmente já tem muita coisa e você só precisa acrescentar. Já a variável **CLASSPATH** deve ser criada. No Linux, são as mesmas variáveis, mas o **PATH** é separado por :. Nos Windows velhos, como o 98, você deve alterar isso no **autoexec.bat**. Nos Windows mais novos, como NT, 2000, e XP, procure onde você pode adicionar novas variáveis de ambiente (em Iniciar - Painel de Controle – Sistema – Avançado Variáveis de Sistema).

Se você possui dúvidas sobre a instalação e configuração geral do ambiente, consulte o tutorial no site do guj: http://www.guj.com.br .

Versões do Java

Existe uma quantidade assombrosa de siglas e números ao redor do Java. No começo isso pode ser bastante confuso, ainda mais porque cada biblioteca do Java mantém seu próprio versionamento.

Talvez, o que seja mais estranho é o termo "Java 2". Sempre que você for ler alguma coisa sobre Java, vai ouvir falar em Java2 ou J2 como prefixo de alguma sigla. Na verdade não existe Java 2.0, acontece que quando a Sun lançou a versão 1.2 do Java fizeram uma jogada de marketing e decidiram chamá-la de Java 2.

Hoje em dia, o Java está na versão 1.5, mas o marketing utiliza "Java2 5.0".

Java 5.0 e Java 1.4

Muitas pessoas estão migrando para o Java 5.0, mas como ele é muito novo, algumas empresas vão se prender ao Java 1.4 durante muito tempo. Houve uma mudança significativa na linguagem entre essas duas versões, com certeza a mais significativa).

No decorrer do curso, todos os recursos e classes que forem exclusivamente do Java2 5.0 terão este fato destacado.

A versão Java 6.0 já está em desenvolvimento e com provável lançamento em meados de 2006. Apesar do nome, não há mudança na linguagem prevista, apenas melhorias na JVM e novas bibliotecas. Além disso o número 2 da sigla ira cair: Java2 de volta para Java. Confuso não?

J2EE?

Se você está começando agora com Java, não deverá começar pelo **J2EE**. Isso não importa agora.

Quando você ouvir falar em **Servlets**, **JSP** e **EJB**, isso tudo faz parte do **J2EE**. Apesar da esmagadora quantidade de vagas de emprego para Java estarem no **J2EE**, ela é apenas uma especificação, algo relativamente simples de aprender depois que você firmar bem os conceitos do Java.

Novamente, não comece aprendendo Java através do **J2EE**.

2.4 - Compilando o primeiro programa

Vamos para o nosso primeiro código! O programa que imprime uma linha simples!

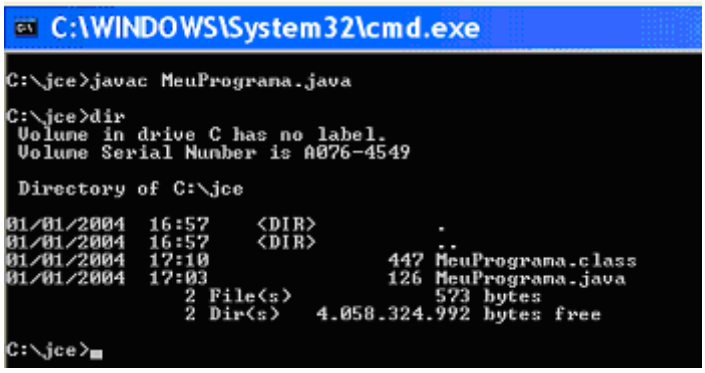
```
1.  class MeuPrograma {
2.    public static void main(String[] args) {
3.          System.out.println("Minha primeira aplicação Java!!");
4.    }
5.  }
```

Notação

Todos os códigos apresentados na apostila estão formatados com recursos visuais para auxiliar a leitura e compreensão dos mesmos. Quando for digitar os códigos no computador, trate os códigos como texto simples.

A numeração das linhas **não** faz parte do código e não deve ser digitada; é apenas um recurso didático. O java é case sensitive: tome cuidado com maiusculas e minusculas.

Após digitar o código acima, grave-o como MeuPrograma.java em algum diretório. Para compilar, você deve pedir para que o compilador de Java da Sun, chamado javac, gere o bytecode correspondente do seu código Java.



Depois de compilar, o **bytecode** foi gerado. Quando o sistema operacional listar os arquivos contidos no diretório atual, você poderá ver que um arquivo .**class** foi gerado, com o mesmo nome da sua classe Java.

Assustado com o código?

Para quem já tem uma experiência com Java, esse primeiro código é muito simples. Mas se é seu primeiro código em Java, pode ser um pouco traumatizante. Não deixe de ler o prefácio do curso, que deixará você mais tranqüilo.

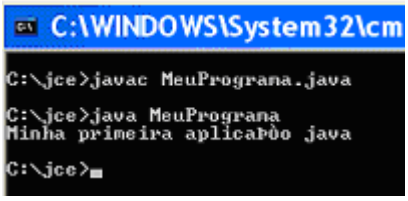
Preciso sempre programar usando o Notepad ou similar?

Não é necessário sempre digitar seu programa em um simples aplicativo como o Notepad. Você pode usar um editor que tenha **syntax highlighting** e outros benefícios.

Mas, no começo, é interessante você usar algo que não possua ferramentas, para que você possa se acostumar com os erros de compilação, sintaxe, e outros. Depois do capítulo de polimorfismo e henraça sugerimos a utilização do Eclipse (www.eclipse.org), a IDE líder do mercado, e gratuita.

2.5 - Executando seu primeiro programa

Os procedimentos para executar seu programa são muito simples. O **javac** é o compilador Java, e o **java** é o responsável por invocar a máquina virtual para interpretar o seu programa.



Ao executar, pode ser que a acentuação resultante saia errada, devido a algumas configurações que deixamos de fazer; sem problemas.

2.6 - O que aconteceu?

```
1.  class MeuPrograma {
2.    public static void main(String[] args) {
3.
4.          // miolo do programa começa aqui!
5.          System.out.println("Minha primeira aplicação Java!!");
6.          // fim do miolo do programa
7.
8.    }
9.  }
```

MAIN

O miolo do programa é o que será executado quando chamamos a máquina virtual. Por enquanto, todas as linhas anteriores, onde há a declaração de uma classe e a de um método, não importa para nós. Mas devemos saber que toda aplicação Java começa por um ponto de entrada, e este ponto de entrada é um

método `main`.

Ainda não sabemos o que é método, mas veremos no capítulo 4. Até lá, não se preocupe com essas declarações. Sempre que um exercício for feito, o código sempre estará nesse miolo.

No caso do nosso código, a linha do `System.out.println` faz com que o conteúdo entre aspas seja colocado na tela.

2.7 - Exercícios

1-) Altere seu programa para imprimir uma mensagem diferente.

2-) Altere seu programa para imprimir duas linhas de texto usando duas linhas de código `System.out`.

3-) Sabendo que os caracteres `\n` representam uma quebra de linhas, imprima duas linhas de texto usando uma única linha de código `System.out`.

2.8 - O que pode dar errado?

Muitos erros podem ocorrer no momento que você rodar seu primeiro código. Vamos ver alguns deles:

Código:

```
public class X {
       public static void main (String[] args) {
              System.out.println("Falta ponto e vírgula")
       }
}
```

Erro:

```
X.java:4: ';' expected
        }
        ^
1 error
```

Esse é o erro de compilação mais comum: aquele onde um ponto e vírgula fora esquecido. Outros erros de compilação podem ocorrer se você escreveu palavras chaves (a que colocamos em negrito) em maiúsculas, esqueceu de abrir e fechar as `{}`, etc.

Durante a execução, outros erros podem aparecer:

- Se você declarar a classe como X, compilá-la e depois tentar usá-la como x minúsculo (java x), o Java te avisa:

```
Exception in thread "main" java.lang.NoClassDefFoundError: X (wrong name: x)
```

- Se tentar acessar uma classe no diretório ou classpath errado, ou se o nome estiver errado, ocorrerá o seguinte erro:

```
Exception in thread "main" java.lang.NoClassDefFoundError: X
```

- Se esquecer de colocar `static` ou o argumento `String[] args` no método `main`:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Por exemplo:

```
public class X {
	public void main (String[] args) {
		System.out.println("Faltou o static");
	}
}
```

- Se não colocar o método `main` como `public`:

```
Main method not public.
```

Por exemplo:

```
public class X {
	static void main (String[] args) {
		System.out.println("Faltou o public");
	}
}
```

2.9 - Um pouco mais...

1-) Procure um colega, ou algum conhecido, que esteja em um projeto Java. Descubra porque Java foi escolhido como tecnologia. O que é importante para esse projeto e o que acabou fazendo do Java a melhor escolha?

2-) Se o software relacionado ao Java que a Sun deixa disponível é gratuito, como a Sun ganha dinheiro?

2.10 - Exercícios

1-) A máquina virtual parece sempre estar traduzindo o seu programa para o sistema operacional em uso, isso faz com que alguns digam que o Java é interpretado. Porém, desde o **Java 1.2**, a máquina virtual faz algo a mais que apenas interpretar: ela compila parte do código que considerar "quente" em tempo de execução. É o que chamamos de compilação dinâmica, pois além disso ela pode mudar a forma de como otimizou aquele trecho de código se não achou o resultado satisfatório. Pesquise sobre isso (as palavras chave são **JIT compiler** e **HotSpot**).

2-) Um arquivo fonte Java deve sempre ter a extensão `.java`, ou o compilador o rejeitará. Além disso, existem algumas outras regras na hora de dar o nome de um arquivo Java. Experimente gravar o código deste capítulo com `OutroNome.java` ou algo similar. Compile e verifique o nome do arquivo gerado. Como executar a sua aplicação agora?

Curiosidade

Tente compilar um arquivo sem nada dentro, nem uma letra, nem uma quebra de linha. O que acontece?

capítulo 3

Variáveis primitivas e Controle de fluxo

"Péssima idéia, a de que não se pode mudar"
Montaigne -

Iremos aprender a trabalhar com os seguintes recursos da linguagem Java:
- declarando, atribuindo valores, casting e comparando variáveis;
- controle de fluxo através de if e else;
- instruções de laço for e while, controle de fluxo com break e continue.

3.1 - Declarando e usando variáveis

VARIÁVEIS

Dentro de um bloco, podemos declarar variáveis e usá-las.

Em Java, toda variável tem um tipo que não pode ser mudado uma vez que declarado:

```
tipoDaVaríavel nomeDaVariável;
```

INT

Por exemplo, é possível ter uma `idade` que vale um **número inteiro**:

```
int idade;
```

Com isso, você declara a variável `idade`, que passa a existir a partir deste momento. Ela é do tipo `int`, que guarda um número inteiro. A partir de agora você pode usá-la, primeiro atribuindo valores.

A linha a seguir é a tradução de **"idade deve valer agora quinze"**.

```
idade = 15;
```

Comentários em Java

Para fazer um comentário em java, você pode usar o `//` para comentar até o final da linha, ou então usar o `/* */` para comentar o que estiver entre eles.

```
/* comentário daqui,
    ate aqui */

// uma linha de comentário sobre a idade
int idade;
```

Além de atribuir, você pode utilizar esse valor. O código a seguir declara novamente a variável idade com valor 15 e imprime seu valor na saída padrão através da chamada a `System.out.println`.

```
// declara a idade
int idade;
idade = 15;
```

```
// imprime a idade
System.out.println(idade);
```

Por fim, podemos utilizar o valor de uma variável para algum outro propósito, como alterar ou definir uma segunda variável. O código a seguir cria uma variável chamada `idadeNoAnoQueVem` com valor de **idade mais um**.

```
// gera uma idade no ano seguinte
int idadeNoAnoQueVem;
idadeNoAnoQueVem = idade + 1;
```

OPERADORES ARITMÉTICOS

Você pode usar os operadores `+`, `-`, `/` e `*` para operar com números, sendo eles responsáveis pela adição, subtração, divisão e multiplicação, respectivamente. Além desses operadores básicos, há o operador `%` (módulo) que nada mais mais é que o **resto de uma divisão inteira**. Veja alguns exemplos:

```
int quatro = 2 + 2;
int tres = 5 - 2;

int oito = 4 * 2;
int dezesseis = 64 / 4;

int um = 5 % 2; // 5 dividido por 2 dá 2 e tem resto 1;
                // o operador % pega o resto da divisão inteira
```

Onde testar esses códigos?

Você deve colocar esses trechos de código dentro do método main, que vimos no capítulo anterior. Isto é, isso deve ficar no miolo do programa. Use bastante `System.out.println`, dessa forma você pode ver algum resultado, caso contrário, ao executar a aplicação, nada aparecerá.

Por exemplo, para imprimir a `idade` e a `idadeNoAnoQueVem` podemos escrever o seguinte programa de exemplo:

```
1.  class TestaIdade {
2.
3.    public static void main(String[] args) {
4.
5.          // declara a idade
6.          int idade;
7.          idade = 15;
8.
9.          // imprime a idade
10.         System.out.println(idade);
11.
12.         // gera uma idade no ano seguinte
13.         int idadeNoAnoQueVem;
14.         idadeNoAnoQueVem = idade + 1;
15.
16.         // imprime a idade
17.         System.out.println(idadeNoAnoQueVem);
18.
19.   }
20. }
```

No momento que você declara uma variável, também é possível inicializá-la por praticidade:

```
int idade = 15;
```

DOUBLE

Representar números inteiros é fácil, mas como guardar valores reais, como frações de números inteiros e outros? Outro tipo de variável muito utilizado é o `double`, que armazena um número com ponto flutuante.

```
double d = 3.14;
double x = 5 * 10;
```

BOOLEAN

O tipo `boolean` armazena um valor verdadeiro ou falso, e só.

```
boolean verdade = true;
```

CHAR

O tipo `char` guarda um e apenas um caractere. Esse caractere deve estar entre aspas simples. Não se esqueça dessas duas características de uma variável do tipo `char`! Por exemplo, ela não pode guardar um código como '' pois o vazio não é um caractere!

```
char letra = 'a';
System.out.println(letra);
```

3.2 - Tipos primitivos e valores

ATRIBUIÇÃO

Esses tipos de variáveis são tipos primitivos do Java: o valor que elas guardam são o real conteúdo da variável. Quando você utilizar o **operador de atribuição** = o valor será **copiado**.

```
int i = 5; // i recebe uma cópia do valor 5
int j = i; // j recebe uma cópia do valor de i
i = i + 1; // i vira 6, j continua 5
```

Aqui, `i` fica com o valor de 6. Mas e `j`? Na segunda linha, `j` está valendo 5. Quando `i` passa a valer 6, será que `j` também fica valendo? Não, pois o valor de um tipo primitivo sempre é copiado.

Apesar da linha 2 fazer `j = i`, a partir desse momento essas variáveis não tem relação nenhuma: o que acontecer com uma não reflete em nada com a outra,

Outros tipos primitivos

Vimos aqui os tipos primitivos que mais aparecem. O Java tem outros, que são o `byte`, `short`, `long` e `float`.

Cada tipo possui características especiais que, para um programador avançado, podem fazer muita diferença.

3.3 - Exercícios

1-) Na empresa onde trabalhamos, há tabelas com o quanto foi gasto em cada mês. Para fechar o balanço do primeiro trimestre, precisamos somar o gasto total. Sabendo que, em Janeiro foi gasto 15000 reais, em Fevereiro, 23000 reais e em Março, 17000 reais, faça um programa que calcule e imprima o gasto total no trimestre. Siga esses passos:

- Crie uma classe chamada BalancoTrimestral com um bloco main, como nos exemplos anteriores;
- Dentro do main (o miolo do programa), declare uma variável inteira chamada gastosJaneiro e inicialize-a com 15000;

- Crie também as variáveis gastosFevereiro e gastosMarco, inicializando-as com 23000 e 17000, respectivamente;
- Crie uma variável chamada gastosTrimestre e inicialize-a com a soma das outras 3 variáveis:
  int gastosTrimestre = gastosJaneiro + gastosFevereiro + gastosMarco
- Imprima a variável gastosTrimestre.

2-) Adicione código (sem alterar as linhas que já existem) no programa a seguir para imprimir o resultado a seguir:

Resultado: 15, 15.1, y, false

```
1.  class ExercicioSimples {
2.
3.    public static void main(String[] args) {
4.
5.          int i = 10;
6.          double d = 5;
7.          char c = 't';
8.          boolean b = true;
9.
10.         // imprime concatenando diversas variáveis
11.         System.out.println("Resultado: " + i + ", " + d + ", " + c + ", " +
    b);
12.
13.   }
14. }
```

3.4 - Casting e promoção

Alguns valores são incompatíveis se você tentar fazer uma atribuição direta. Enquanto um número real costuma ser representado em uma variável do tipo `double`, tentar atribuir ele a uma variável `int` não funciona pois é um código que diz: "**i deve valer d**", mas não se sabe se `d` realmente é um número inteiro ou não.

```
double d = 3.1415;
int i = d; // não compila
```

O mesmo ocorre no seguinte trecho:

```
int i = 3.14;
```

O mais interessante, é que nem mesmo o seguinte código compila:

```
double d = 5; // ok, o double pode conter um número inteiro
int i = d; // não compila
```

Apesar de 5 ser um bom valor para um `int`, o compilador não tem como saber que valor estará dentro desse `double` no momento da execução. Esse valor pode ter sido digitado pelo usuário, e ninguém vai garantir que essa conversão ocorra sem perda de valores.

Já no caso a seguir é o contrário:

```
int i = 5;
double d2 = i;
```

O código acima compila sem problemas, já que um `double` pode guardar um número com ou sem ponto flutuante. Todos os inteiros representados por uma variável do tipo `int` podem ser guardados em uma variável `double`, então não existem problemas no código acima.

CASTING

Ás vezes, precisamos que um número quebrado seja arredondado e armazenado num número inteiro. Para fazer isso sem que haja o erro de compilação, é preciso ordenar que o número quebrado seja **moldado (casted)** como um número inteiro. Esse processo recebe o nome de **casting**.

```
double d3 = 3.14;
int i = (int) d3;
```

O casting foi feito para moldar a variável `d3` como um `int`. O valor dela agora é 3.

O mesmo ocorre entre valores `int` e `long`.

```
long x = 10000;
int i = x; // nao compila, pois pode estar perdendo informação
```

E, se quisermos realmente fazer isso, fazemos o casting:

```
long x = 10000;
int i = (int) x;
```

Casos não tão comuns de casting e atribuição

Alguns **castings** aparecem também:

```
float x = 0.0;
```

O código acima não compila pois todos os literais com ponto flutuante são considerados `double` pelo Java. E `float` não pode receber um `double` sem perda de informação, para fazer isso funcionar podemos escrever o seguinte:

```
float x = 0.0f;
```

A letra `f` indica que aquele literal deve ser tratado como `float`. Outro caso, que é mais comum:

```
double d = 5;
float f = 3;

float x = (float) d + f;
```

Você precisa do casting porque o Java faz as contas e vai armazenando sempre no maior tipo que apareceu durante as operações, no caso o `double`. E no mínimo, o Java armazena em um `int`.

Até casting com variáveis do tipo `char` podem ocorrer. O único tipo primitivo que não pode ser atribuído a nenhum outro tipo é o `boolean`.

Castings possíveis

Abaixo estão relacionados todos os casts possíveis na linguagem Java, mostrando quando você quer converter **de** um valor **para** outro. A indicação **Impl.** quer dizer que aquele cast é implícito e automático, ou seja, você não precisa indicar o cast explicitamente. (lembrando que o tipo boolean não pode ser convertido para nenhum outro tipo)

PARA: / DE:	byte	short	char	int	long	float	double
byte	----	*Impl.*	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
short	(byte)	----	(char)	*Impl.*	*Impl.*	*Impl.*	*Impl.*
char	(byte)	(short)	----	*Impl.*	*Impl.*	*Impl.*	*Impl.*
int	(byte)	(short)	(char)	----	*Impl.*	*Impl.*	*Impl.*
long	(byte)	(short)	(char)	(int)	----	*Impl.*	*Impl.*
float	(byte)	(short)	(char)	(int)	(long)	----	*Impl.*
double	(byte)	(short)	(char)	(int)	(long)	(float)	----

Tamanho dos tipos

Na tabela abaixo, estão os tamanhos de cada tipo primitivo do Java.

TIPO	*TAMANHO*
boolean	1 bit
byte	1 byte
short	2 bytes
char	2 bytes
int	4 bytes
float	4 bytes
long	8 bytes
double	8 bytes

3.5 - O If-Else

IF

A sintaxe do `if` no Java é a seguinte

```
if (condicaoBooleana) {
      codigo;
}
```

CONDIÇÃO BOOLEANA

Uma **condição booleana** é qualquer expressão que retorne `true` ou `false`. Para isso, você pode usar os operadores <, >, <=, >= e outros. Um exemplo:

```
int idade = 15;
if (idade < 18) {
      System.out.println("Não pode entrar");
}
```

ELSE

Além disso, você pode usar a cláusula `else` para indicar o comportamento que deve ser executado no caso da expressão booleana ser falsa:

```
1.  int idade = 15;
2.  if (idade < 18) {
3.     System.out.println("Não pode entrar");
4.  }
5.  else {
6.     System.out.println("Pode entrar");
7.  }
```

Você pode concatenar expressões booleanas através dos operadores lógicos

OPERADORES LÓGICOS

"E" e **"OU"**. O "E" é representado pelo `&` e o "OU" é representado pelo `|`.

```
1.  int idade = 15;
2.  boolean amigoDoDono = true;
3.  if (idade < 18 & amigoDoDono == false) {
4.     System.out.println("Não pode entrar");
5.  }
6.  else {
7.     System.out.println("Pode entrar");
8.  }
```

OPERADOR DE NEGAÇÃO

Esse código poderia ainda ficar mais legível, utilizando-se o operador de negação, o `!`. Esse operador transforma uma expressão booleana de `false` para `true` e vice versa.

```
1.  int idade = 15;
2.  boolean amigoDoDono = true;
3.  if (idade < 18 & !amigoDoDono) {
4.     System.out.println("Não pode entrar");
5.  }
6.  else {
7.     System.out.println("Pode entrar");
8.  }
```

Repare na linha 3 que o trecho `amigoDoDono == false` virou `!amigoDoDono`. **Eles têm o mesmo valor**.

Para comparar se uma variável tem o mesmo valor que outra variável ou valor, utilizamos o operador `==`. Repare que utilizar o operador `=` vai retornar um erro de compilação, já que o operador `=` é o de atribuição.

```
int mes = 1;
if (mes == 1) {
       System.out.println("Você deveria estar de férias");
}
```

&& ou &?

Em alguns livros, logo será apresentado a você dois tipos de operadores de OU e de E. Você realmente não precisa saber distinguir a diferença entre eles por enquanto.

O que acontece é que os operadores `&&` e `||` funcionam como seus operadores irmãos, porém eles funcionam da maneira mais rápida possível, quando percebem que a resposta não mudará mais, eles param de verificar as outras condições booleanas. Por isso eles são chamados de operadores de curto circuito (short circuit operators).

```
if (true | algumaCoisa) {
       // ...
}
```

O valor de `algumaCoisa` será analisado nesse caso. Repare que não precisaria, pois já temos um `true`. *`true` ou qualquer outra coisa* dá sempre `true`.

```
if (true || algumaCoisa) {
       // ...
}
```

Neste caso o `algumaCoisa` não será analisado. Pode não fazer sentido ter as duas opções, mas em alguns casos é interessante e útil usar um ou outro, além de dar diferença no resultado. Veremos mais adiante em outros capítulos.

3.6 - O While

LAÇO

WHILE

O `while` é um comando usado para fazer um **laço** (**loop**), isto é, repetir um trecho de código algumas vezes. A idéia é que esse trecho de código seja repetido enquanto uma determinada condição permanecer verdadeira.

```
int idade = 15;
while(idade < 18) {
      // espera ele crescer
      idade = idade + 1;
}
```

O trecho dentro do bloco do `while` será executado até o momento em que a condição `idade < 18` passe a ser falsa. E isso ocorrerá exatamente no momento em que `idade == 18`, o que fará imprimir `18`.

3.7 - O For

FOR

Outro comando de **loop** extremamente utilizado é o `for`. A idéia é a mesma do `while`, fazer um trecho de código ser repetido enquanto uma condição continuar verdadeira. Mas além disso, o `for` isola também um espaço para inicialização de variáveis e o modificador dessas variáveis. Isso faz com que fique mais legível as variáveis que são relacionadas ao loop:

```
for (inicializacao; condicao; incremento) {
      codigo;
}
```

Um exemplo é o a seguir:

```
for (int i = 0; i < 10; i = i + 1) {
      System.out.println("olá!");
}
```

Repare que esse for poderia ser trocado por:

```
int i = 0;
while (i < 10) {
      i = i + 1;
      System.out.println("olá!");
}
```

Porém, o código do `for` indica claramente que a variável `i` serve em especial para controlar a quantidade de laços executados. Quando usar o `for`? Quando usar o `while`? Depende do gosto e da ocasião.

pós incremento ++

`i = i + 1` pode realmente ser substituído por `i++` quando isolado, porém, em alguns casos, temos o seguinte:

```
int i = 5;
int x = i++;
```

Qual é o valor de `x`? O de `i`, após essa linha, é 6.

O operador `++`, quando vem a frente da variável, retorna o valor antigo, e incrementa (pós incremento), fazendo `x` valer 5.

Se você tivesse usado o ++ antes da variável (pré incremento), o resultado seria 6, como segue:

```
int i = 5;
int x = ++i;
```

3.8 - Exercícios

1-) Imprima todos os números de 150 a 1500 (inclusive).

2-) Imprima a soma de 1 até 1000 (inclusive).

3-) Imprima todos os múltiplos de 3, entre 1 e 100 (inclusive).

4-) Imprima todos os números entre 1 e 100 (inclusive) dizendo se ele é ou não é múltiplo de 3. (dica: use o operador de resto: %)

3.9 - Controlando loops

Apesar de termos condições booleanas nos nossos laços, em algum momento podemos decidir parar o loop por algum motivo especial, sem que o resto do laço seja executado.

```
1.  for (int i = x; i < y; i++) {
2.     if (i % 19 == 0) {
3.           System.out.println("Achei um número divisível por 19 entre x e y");
4.           break;
5.     }
6.  }
```

BREAK

O código acima vai percorrer os números de x a y e parar quando encontrar um número divisível por 19, uma vez que foi utilizada a palavra chave `break`.

CONTINUE

Da mesma maneira, é possível obrigar o loop a executar o próximo laço. Para isso usamos a palavra chave `continue`.

```
1.  for (int i = 0; i < 100; i++) {
2.     if(i > 50 && i < 60) {
3.           continue;
4.     }
5.     System.out.println(i);
6.  }
```

O código acima não vai imprimir alguns números. (Quais exatamente?)

3.10 - Escopo das variáveis

No Java, podemos declarar variáveis a qualquer momento. Porém, dependendo de onde você as declarou, ela vai valer de um determinado ponto a outro.

```
// aqui a variável i não existe
int i = 5;
// a partir daqui ela existe
```

ESCOPO

O **escopo da variável** é o nome dado ao trecho de código em que aquela variável existe e que é possível acessá-la.

Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro **só valem até o fim daquele bloco**.

```
1.  // aqui a variável i não existe
2.  int i = 5;
3.  // a partir daqui ela existe
4.  while (condicao) {
5.     // o i ainda vale aqui
6.     int j = 7;
7.     // o j passa a existir
8.  }
9.  // aqui o j não existe mais, mas o i continua a valer
```

No bloco acima, a variável `j` pára de existir quando termina o bloco onde ela foi declarada. Se você tentar acessar uma variável fora de seu escopo, ocorrerá um erro de compilação.

3.11 - Um bloco dentro do outro

Um bloco também pode ser declarado dentro de outro. Isto é, um `if` dentro de um `for`, ou um `for` dentro de um `for`, algo como:

```
while (condicao) {
       for (int i = 0; i < 10; i++) {
              // código
       }
}
```

3.12 - Um pouco mais...

1-) Vimos apenas os comandos mais usados para controle de fluxo. O Java ainda possui o `do..while` e o `switch`. Pesquise sobre eles e diga quando é interessante usar cada um deles.

2-) Algumas vezes temos vários laços encadeados. Podemos utilizar o `break` para quebrar o laço mais interno, mas se quisermos quebrar um laço mais externo, teremos de encadear diversos `if`s e seu código ficará uma bagunça. O Java possui um artifício chamado **labeled loops**, pesquise sobre eles.

3-) O que acontece se você tentar dividir um número inteiro por 0? E por 0.0?

4-) Existe um caminho entre os tipos primitivos que indicam se há a necessidade ou não de casting entre os tipos. Por exemplo, `int` -> `long` -> `double` (um `int` pode ser tratado como um `double`, mas não o contrário). Pesquise (ou teste), e posicione os outros tipos primitivos nesse fluxo.

5-) Existem outros operadores, como o `%`, `<<`, `>>`. Descubra para que servem.

6-) Além dos operadores de incremento, existem os de decremento, como `--i` e `i--`, além desse, você pode usar instruções do tipo `i += x` e `i -= x`, o que essas instruções fazer? Teste.

3.13 - Exercícios

1-) Imprima todos os múltiplos de 3 ou de 5, entre 1 e 100 (inclusive):

- Dentro do método `main`, faça um `for` que varie de 1 a 100:
  `for (int i = 1; i <= 100; i++)`
- Em cada iteração, verifique se `i` é múltiplo de 3 ou 5 (use o operador % para isso):
  `if (i % 3 == 0 || i % 5 == 0)`
- Se for, múltiplo, imprima o número

2-) Imprima os fatoriais de 1 a 10.

O fatorial de um número n é n * n-1 * n-2 ... até n = 1. Lembre-se de utilizar os parênteses.

O fatorial de 0 é 1
O fatorial de 1 é (0!) * 1 = 1
O fatorial de 2 é (1!) * 2 = 2
O fatorial de 3 é (2!) * 3 = 6
O fatorial de 4 é (3!) * 4 = 24

- Faça um for que inicie uma variável n (número) como 1 e fatorial (resultado) como 1 e varia n de 1 até 10:

```
for (int n=1, fatorial=1; n <= 10; n++) {
```

3-) Aumente a quantidade de números que terão os fatoriais impressos, até 20, 30, 40. Em um determinado momento, além desse cálculo demorar, vai começar a mostrar respostas completamente erradas. Porque? Mude de `int` para `long`, e você poderá ver alguma mudança.

4-) Imprima os primeiros números da série de Fibonacci até passar de 100. A série de Fibonacci é a seguinte: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc... Para calculá-la, o primeiro e segundo elementos valem 1, daí por diante, o n-ésimo elemento vale o n-1-ésimo elemento somando ao n-2-ésimo elemento (ex: 8 = 5 + 3).

5-) Escreva um programa que, dada uma variável `x` (com valor 180, por exemplo), temos `y` de acordo com a seguinte regra:

se x é par, y = x / 2
se x é impar, y = 3 * x + 1
imprime y

O programa deve então jogar o valor de y em x e continuar até que y tenha o valor final de 1. Por exemplo, para x = 13, a saída será:

40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1

Imprimindo sem pular linha

Um detalhe importante do método que estamos usando até agora é que uma quebra de linha é impressa toda vez que chamado. Para não pular uma linha usamos o método a seguir:

```
System.out.print(variavel);
```

6-) Imprima a seguinte tabela, usando `for`s encadeados:

1
2 4
3 6 9
4 8 12 16
n n*2 n*3 .... n*n

3.14 - Desafios

1-) Faça o exercício da série de Fibonacci usando apenas duas variáveis.

2-) Imprima o triângulo de pascal até a n-ésima linha (deixe uma variável `int linha` para que você possa escolher quantas linhas deseja imprimir).

capítulo 4

Orientação a objetos básica

"Programação orientada à objetos é uma péssima idéia, que só poderia ter nascido na Califórnia."
Edsger Dijkstra -

Ao término deste capítulo, você será capaz de:

- dizer o que é e para que serve orientação a objetos,
- conceituar classes, atributos e comportamentos e
- entender o significado de variáveis e objetos na memória.

4.1 - O problema

ORIENTAÇÃO À OBJETOS

Orientação à objetos é uma maneira de programar que ajuda na organização e resolve muitos problemas enfrentados pela programação procedural.

Consideremos o clássico problema da validação de um CPF. Normalmente, temos um formulário, no qual recebemos essa informação, e depois temos que enviar esses caracteres para uma função que irá validá-lo.

Alguém te obriga a sempre validar esse CPF?

Você pode, inúmeras vezes, esquecer de chamar esse validador.

Considerando que você não erre aí, ainda temos outro problema: imagine que em algum caso, você não vá validar o CPF, ou valide de outra maneira. Por exemplo, queremos validar o CPF apenas das pessoas maiores que 18 anos. Vamos ter de colocar um `if`... mas onde? Espalhado por todo seu código...

A responsabilidade de estar verificando se o cliente tem ou não tem 18 anos, ficou espalhada por todo seu código. Seria legal poder concentrar essa responsabilidade em um lugar só, para não ter chances de esquecer isso.

Não só por isso, imagine que em algum momento precisaremos mudar essa condição... vai ter novamente de procurar todos os ifs do seu código!

Não existe uma conexão entre seus dados! Não existe uma conexão entre seus dados e suas funcionalidades! A idéia é ter essa amarra através da linguagem.

Quais as vantagens?

Orientação a objetos vai te ajudar em muito em se organizar e escrever menos, além de concentrar as responsabilidades nos pontos certos, flexibilizando sua aplicação.
Outra enorme vantagem, de onde você realmente vai economizar montanhas de código, é o **polimorfismo**, que veremos em um posterior capítulo.

4.2 - Criando um tipo

Considere um programa para um banco, é bem fácil perceber que uma entidade extremamente importante para o nosso sistema é a conta. Nossa idéia aqui é generalizarmos alguma informação, juntamente com funcionalidades que toda conta deve ter.

O que toda conta tem e é importante para nós?

- número da conta
- nome do cliente
- saldo
- limite

O que toda conta faz e é importante para nós? Isto é, o que gostaríamos de "pedir à conta".

- saca uma quantidade x
- deposita uma quantidade x
- imprime o nome do dono da conta
- devolve o saldo atual
- transfere uma quantidade x para uma outra conta y
- devolve o tipo de conta

Com isso temos o projeto de uma conta bancária. Podemos pegar esse projeto e acessar seu saldo? Não. O que temos ainda é o **projeto**. Antes precisamos **construir** uma cconta, para poder acessar o que ela tem, e pedir para ela fazer alguma coisa.

CLASSE

Ao projeto da conta, isto é, a definição da conta, damos o nome de **classe**. O que podemos construir a partir desse projeto, que são as contas de verdade, damos o nome de **objetos**.

4.3 - Uma classe em Java

Um tipo desses, como especificado acima, pode ser facilmente traduzido para Java:

```
1.  class Conta {
2.     int numero;
3.     String dono;
4.     double saldo;
5.     double limite;
6.
7.     // ..
8.
9.  }
```

String

String é uma classe em Java. Ela guarda uma palavra, isso é um punhado de caracteres. Como estamos aprendendo o que é uma classe, entenderemos melhor mais para frente.

ATRIBUTO

Por enquanto declaramos o que toda conta deve ter. Estes são os **atributos** que toda conta, quando criada, vai ter. Repare que essas variáveis foram declaradas fora de um bloco, diferente do que a gente fazia quando tinha aquele `main`. Quando uma variável é declarada diretamente dentro do escopo da classe, é chamada de variável de objeto, ou atributo.

MÉTODO

Dentro da classe, também iremos declarar o que cada conta faz, e como isto é feito. Os comportamentos que cada classe tem, isto é, o que ela faz, é chamado de **método**. Vamos começar por um simples:

```
void imprimeBanco() {
      System.out.println("Esta conta é do Banco J.");
}
```

VOID

A palavra chave `void` diz que, quando você pedir para a conta imprimir o nome do banco, nenhuma informação será enviada de volta a quem pediu.

Outra opção é sacar algum dinheiro:

```
void saca(double quantidade) {
      double novoSaldo = this.saldo - quantidade;
      this.saldo = novoSaldo;
}
```

ARGUMENTO

PARÂMETRO

Aqui acontecem várias coisas. Em primeiro lugar, quando alguém pedir para sacar, ele também vai dizer quanto quer sacar. Por isso precisamos declarar o método com algo dentro dos parênteses, o que vai aí dentro é chamado de **argumento** do método (ou **parâmetro**). Essa variável é uma variável comum, chamada também de temporária, pois ao final da execução desse método, ela deixa de existir.

THIS

Dentro do método, estamos declarando uma nova variável. Essa variável, assim como o argumento, vai morrer no fim do método, pois este é seu escopo. No momento que vamos acessar nosso atributo, usamos a palavra chave `this` para mostrar que esse é um atributo, e não uma simples variável.

Repare que nesse caso, a conta pode estourar o limite fixado pelo banco. Mais para frente iremos evitar essa situação, e de uma maneira muito elegante.

Da mesma forma, temos o método para depositar alguma quantia:

```
void deposita(double quantidade) {
      this.saldo += quantidade;
}
```

Observe que, agora, não usamos uma variável auxiliar e ainda usamos a abreviação `+=` para deixar o método bem simples.

4.4 - Criando e usando um objeto

Agora temos uma classe em Java, que especifica o que toda classe deve ter. Mas como usá-la? Além dessa, ainda teremos o **Programa.java**, e a partir dele é que iremos acessar a classe `Conta`.

NEW

Para criar (construir, instanciar) uma conta, basta usar a palavra chave `new`:

```
1.  class Programa {
2.    public static void main(String[] args) {
3.          Conta minhaConta;
4.          minhaConta = new Conta();
5.          // …
6.    }
7.  }
```

Através da variável `minhaConta` podemos acessar o objeto recém criado para alterar seu `dono`, seu `saldo` etc:

```
minhaConta.dono = "Duke";
minhaConta.saldo = 1000.0;
minhaConta.limite = 3000.0;
```

INVOCAÇÃO DE MÉTODO

É importante fixar que o ponto foi utilizado para acessar algo em `minhaConta`. Agora, `minhaConta` pertence ao Duke, tem saldo de mil reais e limite de 3 mil reais. Para mandar uma mensagem ao objeto, e pedir que ele execute um método, também usamos o ponto. O termo usado para isso é uma **invocação de método**.

O código a seguir saca um dinheiro e depois deposita outra quantia na nossa conta:

```
1.  class SacaEDeposita {
2.    public static void main(String[] args) {
3.          // criando a conta
4.          Conta minhaConta;
5.          minhaConta = new Conta();
6.
7.          // alterando os valores de minhaConta
8.          minhaConta.dono = "Duke";
9.          minhaConta.saldo = 1000;
10.         minhaConta.limite = 3000;
11.
12.         // saca 200 reais
13.         minhaConta.saca(200);
14.
15.         // deposita 500 reais
16.         minhaConta.deposita(500);
17.         System.out.println(minhaConta.saldo);
18.   }
19. }
```

Uma vez que seu saldo inicial é mil reais, se sacamos 200 reais, depositamos 500 reais e imprimimos o valor do saldo, o que será impresso?

4.5 - Métodos com retorno

Também temos um outro tipo de método, aquele que devolve o tipo de conta com base no limite:

```
boolean saca(double valor) {
      if (this.saldo < valor) {
            return false;
      }
      else {
            this.saldo = this.saldo - valor;
            return true;
      }
}
```

RETURN

Agora a declaração do método mudou! O método `saca` não tem `void` na frente, isto quer dizer que, quando acessado, ele devolve algum tipo de informação. No caso, um boolean. A palavra chave `return` indica que o método vai terminar ali, retornando tal informação.

Exemplo de uso:

```
minhaConta.saldo = 1000;
boolean consegui = minhaConta.saca(2000);
System.out.println(consegui);
```

Ou então posso eliminar a variável temporária, se desejado:

```
minhaConta.saldo = 1000;
System.out.println(minhaConta.saca(2000));
```

Meu programa pode manter na memória não só uma conta, como mais de uma:

```
Conta meuSonho;
meuSonho = new Conta();
meuSonho.saldo = 1500000;
meuSonho.limite = 1000000;
meuSonho.saca(25000);
```

4.6 - O método transfere()

E se quisermos ter um método que `transfere` dinheiro entre duas contas? Podemos fica tentados a criar um método que recebe dois parâmetros: `conta1` e `conta2` do tipo `Conta`. Mas cuidado: assim estamos pensando de maneira procedural.

A idéia é que quando chamarmos o método transfere, já teremos um objeto do tipo `Conta`, portanto o método recebe apenas **um** parâmetro do tipo, a `Conta` destino (além do valor):

```
class Conta {

	// atributos e metodos...

	void transfere(Conta destino, double valor) {
		this.saldo = this.saldo - valor;
		destino.saldo = destino.saldo + valor;
	}
}
```

Para deixar o código mais robusto, poderíamos verificar se a conta possui a quantidade a ser transferida disponível. Para ficar ainda mais interessante, você pode chamar os métodos `deposita` e `saca` já existentes para fazer essa tarefa:

```
class Conta {

	// atributos e metodos...

	boolean transfere(Conta destino, double valor) {
		boolean retirou = this.saca(valor);
		if (retirou == false) {
			// não deu pra sacar!
			return false;
		}
		else {
			destino.deposita(valor);
			return true;
		}
	}
}
```

Esse código poderia ser escrito com uma sintaxe muito mais sucinta. Como?

4.7 - Objetos são acessados por variáveis referências!

REFERÊNCIA

Quando declaramos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência**.

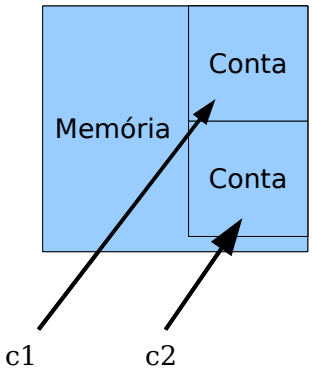
```
1. public static void main(String args[]) {
2.     Conta c1;
3.     c1 = new Conta();
4.     Conta c2;
```

```
5.      c2 = new Conta();
6. }
```

O correto aqui é dizer que `c1` se refere a um objeto. Não é correto dizer que `c1` é um objeto, pois `c1` é uma variável referência.

Temos agora seguinte situação:

```
Conta c1;
c1 = new Conta();

// ...

Conta c2;
c2 = new Conta();
```

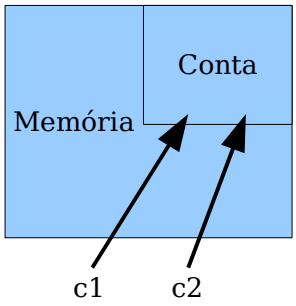



```
1.    public static void main(String args[]) {
2.     Conta c1 = new Conta();
3.     Conta c2 = c1;
4.
5.     c1.saldo = 23;
6.     System.out.println(c2.saldo);
7.    }
```

O que acontece aqui? O operador `=` copia o valor de uma variável. Mas qual é o valor da variável `c1`? É o objeto? Não. Na verdade, o valor guardado é a referência (**endereço**) para onde o objeto se encontra na memória principal.

Na memória, o que acontece nesse caso:

```
Conta c1 = new Conta();
Conta c2 = c1;
```



Quando fizemos `c2 = c1`, `c2` passa a fazer referência para o mesmo objeto que `c1` referencia nesse instante.

new

O que exatamente faz o new?

O new executa uma série de tarefas, que veremos mais adiante.

Mas, para melhor entender as referências no Java, saiba que o new, depois de alocar a memória para esse objeto, devolve uma "flecha", isto é, um valor de referência. Quando você atribui isso em uma variável, essa variável passa a se referir para esse mesmo objeto.

Podemos então ver outra situação:

```
1.    public static void main(String args[]) {
2.     Conta c1 = new Conta();
3.     c1.dono = "Duke";
4.     c1.saldo = 227;
5.
6.     Conta c2 = new Conta();
7.     c2.dono = "Duke";
8.     c2.saldo = 227;
9.
10.    if (c1 == c2) {
11.          System.out.println("Contas iguais");
12.    }
13.   }
```

O operador `==` compara o conteúdo das variáveis, mas essas variáveis não guardam o objeto, e sim o endereço em que ele se encontra. Como em cada uma dessas variáveis guardamos duas contas criadas diferentemente, eles estão em espaços diferentes da memória, o que faz o teste no `if` valer `false`. As contas podem ser equivalentes no nosso critério de igualdade, porém eles não são o mesmo. Quando se trata de objetos, pode ficar mais fácil pensar que o `==` compara se os objetos (referências na verdade) são o mesmo, e não se são iguais?

Para saber se dois objetos tem o mesmo conteúdo, você precisa comparar atributo por atributo. Veremos uma solução mais elegante para isso também.

4.8 - Continuando com atributos

As variáveis do tipo atributo, diferentemente das variáveis temporárias (declaradas dentro de um método), recebem um valor padrão. No caso numérico, valem 0, no caso de `boolean`, vale `false`.

VALORES DEFAULT

Você também pode dar **valores default**, como segue:

```
1.    class Conta {
2.     int numero = 1234;
3.     String dono = "Duke";
4.     String cpf = "123.456.789-10";
5.     double saldo = 1000;
6.     double limite = 1000;
7.    }
```

Nesse caso, quando você criar um carro, seus atributos já estão "populados" com esses valores colocados.

Seus atributos também podem ser referências para outras classes. Suponha a seguinte classe `Cliente`:

```
1.    class Cliente {
2.     String nome;
3.     String sobrenome;
4.     String cpf;
5.    }
6.
7.    class Conta {
8.     int numero;
9.     double saldo;
10.    double limite;
11.    Cliente cliente;
12.    // ..
```

```
13.   }
```

E dentro do `main` da classe de teste:

```
1.    class Teste {
2.     public static void main(String[] args) {
3.           Conta minhaConta = new Conta();
4.           Cliente c = new Cliente();
5.           minhaConta.cliente = c;
6.           // ...
7.     }
8.    }
```

Aqui simplesmente houve uma atribuição. O valor da variável `c` é copiado para o atributo `cliente` do objeto a qual `minhaConta` se refere. Em outras palavras, `minhaConta` agora tem uma referência ao mesmo `Cliente` que `c` se refere, e pode ser acessado através de `minhaConta.cliente`.

Você pode realmente navegar sobre toda essa estrutura de informação, sempre usando o ponto:

```
Cliente clienteDaMinhaConta = minhaConta.cliente;
clienteDaMinhaConta.nome = "Duke";
```

Ou ainda pode fazer isso de uma forma mais direta, e até mais elegante:

```
minhaConta.cliente.nome = "Duke";
```

NULL

Mas e se dentro do meu código eu não desse new em Cliente e tentasse acessa-lo diretamente?

```
1.    class Teste {
2.     public static void main(String[] args) {
3.           Conta minhaConta = new Conta();
4.
5.           minhaConta.cliente.nome = "paulo";
6.           // ...
7.     }
8.    }
```

Quando damos `new` em um objeto, ele o incializa com seus valores default, 0 para números, `false` para `boolean` e `null` para referências. `null` é uma palavra chave em java, que indica uma referência para nenhum objeto.

Se em algum caso você tentar acessar um atributo ou método de alguém que está se referenciando para `null`, você receberá um erro durante a execução (`NullPointerException`, que veremos mais a frente). Da para perceber então que o `new` não traz um efeito cascata, a menos que você de um valor default (ou use construtores que também veremos mais a frente):

```
1.
2.    class Conta {
3.     int numero;
4.     double saldo;
5.     double limite;
6.     Cliente cliente = new Cliente(); // quando chamarem new Conta,
7.                                      //havera um new Cliente para ele.
8.    }
```

Com esse código, toda nova `Conta` criada criado já terá um novo `Cliente` associado, sem necessidade de instanciá-lo logo em seguida da instanciação de uma `Conta`. Qual alternativa você deve usar? Depende do caso: para toda nova `Conta`

você precisa de um novo `Cliente`? É essa pergunta que deve ser respondida. Nesse nosso caso a resposta é não, mas depende do nosso problema.

4.9 - Uma Fábrica de Carros

Além do Banco que estamos criando, vamos ver como ficariam certas classes relacionadas à uma fábrica de carros. Vamos criar uma classe `Carro`, com certos atributos que descrevem suas características e com certos métodos que descrevem seu comportamento.

```
1.   class Carro {
2.    String cor;
3.    String modelo;
4.    double velocidadeAtual;
5.    double velocidadeMaxima;
6.
7.    //liga o carro
8.    void liga() {
9.          System.out.println("O carro está ligado");
10.   }
11.
12.   //acelera uma certa quantidade
13.   void acelera(double quantidade) {
14.         double velocidadeNova = this.velocidadeAtual + quantidade;
15.         this.velocidadeAtual = velocidadeNova;
16.   }
17.
18.   //devolve a marcha do carro
19.   int pegaMarcha() {
20.         if (this.velocidadeAtual < 0) {
21.               return -1;
22.         }
23.         if (this.velocidadeAtual >= 0 && this.velocidadeAtual < 40) {
24.               return 1;
25.         }
26.         if (this.velocidadeAtual >= 40 && this.velocidadeAtual < 80 {
27.               return 2;
28.         }
29.         return 3;
30.   }
31.  }
```

Agora, vamos testar nosso Carro em um programa de testes:

```
1.   class TestaCarro {
2.    public static void main(String[] args) {
3.          Carro meuCarro;
4.          meuCarro = new Carro();
5.          meuCarro.cor = "Verde";
6.          meuCarro.modelo = "Fusca";
7.          meuCarro.velocidadeAtual = 0;
8.          meuCarro.velocidadeMaxima = 80;
9.
10.         // liga o carro
11.         meuCarro.liga();
12.
13.         // acelera o carro
14.         meuCarro.acelera(20);
15.         System.out.println(meuCarro.velocidadeAtual);
16.   }
17.  }
```

Nosso carro pode conter também um Motor:

```
1.   class Motor {
2.    int potencia;
3.    String tipo;
4.   }
```

```
5.
6.    class Carro {
7.     String cor;
8.     String modelo;
9.     double velocidadeAtual;
10.    double velocidadeMaxima;
11.    Motor motor;
12.
13.    // ..
14.   }
```

Podemos agora criar diveros Carros e mexer com seus atributos e métodos, assim como fizemos no exemplo do Banco.

4.10 - Um pouco mais...

1) Quando declaramos uma classe, um método, ou um atributo, podemos dar o nome que quiser, seguindo uma regra. Por exemplo, o nome de um método não pode começar com um número. Pesquise sobre essas regras.

2-) Como você pode ter reparado, sempre damos nomes as variáveis com letras minúsculas. É que existem **convenções de código**, dadas pela Sun, para facilitar a legibilidade do código entre programadores. Essa convenção é *muito seguida*. Pesquise sobre ela no http://java.sun.com, procure por "code conventions".

3-) É necessário usar a palavra chave `this` quando for acessar um atributo? Para que então utilizá-la?

4-) O exercício a seguir irá pedir para modelar um "funcionário". Existe um padrão para representar suas classes em diagramas que é amplamente utilizado chamado **UML**. Pesquise sobre ele.

4.11 - Exercícios

O modelo de funcionários a seguir será utilizado para os exercícios no futuro.
É extremamente importante que o aluno faça esses exercícios para acompanhar o que é dado em aula.

O objetivo aqui é criar um sistema para gerenciar os funcionários do Banco J. Os exercícios desse capítulo são extremamente importantes.

1-) Modele um funcionário. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário, a data de entrada no banco (`String`), seu RG (`String`), e um valor booleano que indique se o funcionário está na empresa no momento ou se já foi embora.

Você deve criar alguns métodos de acordo com o que você sentir necessidade. Além deles, crie um método `bonifica` que aumenta o `salario` do funcionário de acordo com o parâmetro passado como argumento. Crie também um método `demite` que não recebe parâmetro algum, só modifica o valor booleano indicando que o funcionário não trabalha mais aqui.

A idéia aqui é apenas modelar, isto é, só identifique que informações são importantes, e o que um funcionário faz.

2-) Transforme o modelo acima em uma classe Java. Teste-a, usando uma outra classe que tenha o `main`. Você deve criar a classe do funcionário chamada `Funcionario`, e a classe de teste você pode nomear como quiser. A de teste deve possuir o método `main`.

Um esboço da classe:

```
class Funcionario {

	double salario;

	// seus outros atributos e métodos

	void bonifica(double valor) {
		// o que fazer aqui dentro?
	}
}
```

Lembre-se de seguir a conveção java, isso é importantíssimo. Isto é, `nomeDeAtributo`, `nomeDeMetodo`, `nomeDeVariavel`, `NomeDeClasse`, etc...

Todas as classes no mesmo arquivo?

Por enquanto, você pode colocar todas as classes no mesmo arquivo, e apenas compile esse arquivo. Ele vai gerar os dois .class.

Porém é boa prática criar um arquivo .java para cada classe, e em determinados casos, você será obrigado a declarar uma classe em um arquivo separado, como veremos no capítulo 10. Isto não é importante para o aprendizado no momento.

3-) Crie um método `mostra`(), que não recebe nem devolve parâmetro algum, e simplesmente imprime todos os atributos do nosso funcionário. Dessa maneira você não precisa ficar copiando e colando um monte de `System.out.println`() para cada mudança e teste que fizer com cada um de seus funcionários, você simplesmente vai fazer:

```
Funcionario f1 = new Funcionario():
// brincadeiras com f1....
f1.mostra();
```

Veremos mais a frente o método `toString`, que é uma solução muito mais elegante para mostrar a representação de um objeto como `String`, além de não jogar tudo pro System.out (só se você desejar).

O esqueleto do método ficaria assim:

```
class Funcionario {

	// seus outros atributos e métodos

	void mostra() {
		System.out.println("Nome: " + this.nome);
		// imprimir aqui os outros atributos...
	}
}
```

4-) Construa dois funcionários com o `new`, e compare-os com o `==`. E se eles tiverem os mesmos atributos?

5-) Crie duas referências para o **mesmo** funcionário, compare-os com o `==`. Tire suas conclusões. Para criar duas referências pro mesmo funcionário:

```
Funcionario f1 = new Funcionario():
Funcionario f2 = f1;
```

6-) Em vez de utilizar uma `String` para representar a data, crie uma outra classe, chamada Data. Ela possui 3 campos int, para dia, mês e ano. Faça com que

seu funcionário passe a usá-la. (é parecido com o último exemplo, que a `Conta` passou a ter referência para um `Cliente`).

7-) O que acontece se você tentar acessar um atributo diretamente na classe? Como por exemplo:

```
Conta.saldo = 1234;
```

Esse código faz sentido? E este:

```
Conta.saca(50);
```

Faz sentido pedir para o esquema do conta sacar uma quantia?

4.12 - Desafios

1-) Um método pode chamar ele mesmo. Chamamos isso de **recursão**. Você pode resolver a série de fibonacci usando um método que chama ele mesmo. O objetivo é você criar uma classe, que possa ser usada da seguinte maneira:

```
Fibonacci fibo = new Fibonacci();
int i = fibo.calculaFibonacci(5);
System.out.println(i);
```

Aqui ira imprimir 8, já que este é o sexto número da série.

Este método `calculaFibonacci` não pode ter nenhum laço, só pode chamar ele mesmo como método. Pense nele como uma função, que usa a própria função para calcular o resultado.

2-) Porque o modo acima é extremamente mais lento para calcular a série do que o modo iterativo (que se usa um laço)?

3-) Escreva o método recursivo novamente, usando apenas uma linha. Para isso, pesquise sobre o **operador condicional ternário**. (ternary operator)

4.13 - Fixando o conhecimento

O objetivo dos exercícios a seguir é fixar o conceito de classes e objetos, métodos e atributos. Dada a estrutura de uma classe, basta traduzi-la para a linguagem Java e fazer uso de um objeto da mesma em um programa simples.

Se você está com dificuldade em alguma parte desse capítulo, aproveite e treine tudo o que vimos até agora nos pequenos programas abaixo:

Programa 1

Classe: Pessoa.
Atributos: nome, idade.
Método: void fazAniversario()

Crie uma pessoa, coloque seu nome e idade inicial, faça alguns aniversários (aumentando a idade) e imprima seu nome e sua idade.

Programa 2

Classe: Porta
Atributos: aberta, cor, dimensaoX, dimensaoY, dimensaoZ

Métodos: void abre(), void fecha(),
void pinta(String s), boolean estaAberta()

Crie uma porta, abra e feche a mesma, pinte-a de diversas cores, altere suas dimensões e use o metodo `estaAberta` para verificar se ela esta aberta.

Programa 3

Classe: Casa
Atributos: cor, porta1, porta2, porta3
Método: void pinta(String s),
int quantasPortasEstaoAbertas()

Crie uma casa e pinte-a. Crie três portas e coloque-as na casa; abra e feche as mesmas como desejar. Utilize o método `quantasPortasEstaoAbertas` para imprimir o número de portas abertas.

capítulo 5

Um pouco de arrays

"O homem esquecerá antes a morte do pai que a perda da propriedade"
Maquiavel -

Ao término desse capítulo, você será capaz de:
- declarar e instanciar arrays;
- popular e percorrer arrays.

5.1 - O problema

Dentro de um bloco, podemos declarar variáveis e usá-las.

```
int idade1;
int idade2;
int idade3;
int idade4;
```

MATRIZ
ARRAY

Podemos declarar uma **matriz** (**array**) de inteiros:

```
int[] idades;
```

O `int[]` é um tipo. Uma array é sempre um objeto, portanto, a variável idades é uma referência. Vamos precisar criar um objeto para poder usar a array. Como criamos o objeto-array?

```
idades = new int[10];
```

Aqui o que fizemos foi criar uma array de int de 10 posições, e atribuir o endereço o qual ela foi criada. Agora podemos acessar as posições do array.

```
idades[5] = 10;
```

O código acima altera a sexta posição do array. No Java, os índices do array vão de 0 a n-1, onde n é o tamanho dado no momento que você criou a array. Se você tentar acessar uma posição fora desse alcance, um erro ocorrerá durante a execução.

Arrays – um problema no aprendizado de muitas linguagens

Aprender a usar arrays às vezes pode ser um problema em qualquer linguagem. Isso porque envolve uma série de conceitos, sintaxe, e outros. No Java, muitas vezes utilizamos outros recursos em vez de arrays, em especial os pacotes de coleções do Java, que veremos no capítulo 11. Portanto, fique tranqüilo caso não consiga digerir toda sintaxe das arrays num primeiro momento.

5.2 - Arrays de referências

É comum ouvirmos "array de objetos". Porém quando criamos uma array de

alguma classe, ela possui referências. O objeto, como sempre, esta na memória principal, e na sua array só ficam guardadas as **referências** (endereços).

```
Conta[] minhasContas;
minhasContas = new Conta[10];
```

Quantas contas foram criadas aqui? Na verdade, **nenhuma**. Foram criados 10 espaços que você pode utilizar para guardar uma referência a uma Conta. Por enquanto, eles se referenciam para lugar nenhum (null). Se você tentar:

```
System.out.println(minhasContas[0].saldo);
```

Um erro durante a execução ocorrerá! Pois na primeira posição da array não há uma referência para uma conta, nem para lugar nenhum. Você deve **popular** sua array antes.

```
Conta contaNova = new Conta();
contaNova.saldo = 1000.0;
minhasContas[0] = contaNova;
```

Ou você ainda pode fazer isso diretamente:

```
minhasContas[1] = new Conta();
minhasContas[1].saldo = 3200.0;
```

Uma array de tipos primitivos guarda valores, uma array de objetos guarda referências.

5.3 - Percorrendo uma array

Percorrer uma array é muito simples quando fomos nós que a criamos:

```
1.  public static void main(String args[]) {
2.     int[] idades = new int[10];
3.     for (int i = 0; i < 10; i++) {
4.           idades[i] = i * 10;
5.     }
6.     for (int i = 0; i < 10; i++) {
7.           System.out.println(idades[i]);
8.     }
9.  }
```

Porém, em muitos casos, recebemos uma array como argumento em um método:

```
1.  void imprimeArray(int[] array) {
2.     // não compila!!
3.     for (int i = 0; i < ????; i++) {
4.           System.out.println(array[i]);
5.     }
6.  }
```

Até onde o `for` deve ir? Toda array em Java tem um atributo que se chama `length`, e você pode acessá-lo para saber o tamanho da array a qual você está se referenciando naquele momento:

```
1.  void imprimeArray(int[] array) {
2.     for (int i = 0; i < array.length; i++) {
3.           System.out.println(array[i]);
4.     }
5.  }
```

Arrays não podem mudar de tamanho

A partir do momento que uma array foi criada, ela **não pode** mudar de tamanho.

Se você precisar de mais espaço, será necessário criar uma nova array, e antes de se referenciar para ela, copie os elementos da array velha.

5.4 - Percorrendo uma array no Java 5.0

O Java 5.0 traz uma nova sintaxe para percorremos arrays (e coleções, que veremos mais a frente).

No caso de você não ter necessidade de manter uma variável com o índice que indica a posição do elemento no vetor, podemos usar o **enhanced-for**.

```
1.  public static void main(String args[]) {
2.     int[] idades = new int[10];
3.     for (int i = 0; i < 10; i++) {
4.           idades[i] = i * 10;
5.     }
6.     for (int x : idades) {
7.           System.out.println(x);
8.     }
9.  }
```

E agora nem precisamos mais do length para percorrer matrizes que não conhecemos seu tamanho:

```
1.  void imprimeArray(int[] array) {
2.     for (int x : array) {
3.           System.out.println(x);
4.     }
5.  }
```

5.5 - Um pouco mais...

1-) Arrays podem ter mais de uma dimensão. Isto é, em vez de termos uma array de 10 contas, podemos ter uma array de 10 por 10 contas, e você pode acessar a conta na posição da coluna x e linha y. Na verdade, uma array bidimensional em Java é uma array de arrays. Pesquise sobre isso.

2-) Uma array bidimensional não precisa ser retangular, isto é, cada linha pode ter um número diferente de colunas. Como? Porque?

3-) O que acontece se criar uma array de 0 elementos? e -1?

4-) O método main recebe uma array de Strings como argumento. O que vem nela? Pesquise e teste.

5.6 - Exercícios

1-) Criar uma classe `Empresa`. A `Empresa` tem um nome, cnpj e uma array de `Funcionario`, além de outros atributos que você julgar necessário

```
class Empresa {
      // outros atributos
      Funcionario[] funcionarios;
      String cnpj;
}
```

2-) A empresa deve ter um método `adiciona` que recebe uma referência a `Funcionario` como argumento, e guarda esse funcionario. Algo como:

```
...
    void adiciona(Funcionario f) {
        // ...
    }
...
```

Você deve inserir o `Funcionario` em uma posição da array que esteja livre. Existem várias maneira para você fazer isso: guardar um contador para indicar qual a próxima posição vazia ou procurar por uma posição vazia toda vez. O que seria mais interessante?

É importante reparar que o método adiciona não recebe nome, rg, salário, etc. Essa seria uma maneira nem um pouco estruturada, muito menos orientada a objetos de se trabalhar. Você antes cria um `Funcionario` e já passa a referência dele, que dentro do objeto possui rg, salário, etc.

3-) Crie uma outra classe, que vai possuir o seu método main. Dentro dele crie algumas instâncias de `Funcionario` e passe para a empresa pelo método `adiciona`. Repare que antes você vai precisar criar a array, pois inicialmente o atributo funcionarios da classe `Empresa` não se referencia a lugar nenhum (`null`):

```
Empresa empresa = new Empresa();
empresa.funcionarios = new Funcionario[10];
// ....
```

Ou você pode construir a array dentro da própria declaração da classe `Empresa`.

Crie alguns funcionários e passe como argumento para o adiciona da empresa:

```
Funcionario f1 = new Funcionario();
f1.salario = 1000;
empresa.adiciona(f1);
```

Você pode criar esses funcionários dentro de um loop se preferir.

Opcional: o método `adiciona` pode gerar uma mensagem de erro indicando que a array está cheia.

4-) Percorra o atributo funcionarios da sua instância da `Empresa` e imprima o salários de todos seus funcionários.

Ou você pode chamar o método `mostra`() de cada `Funcionario` da sua array.

Use também o `for` novo do java 5.0.

Cuidado: alguns índices do seu array podem não conter referência para `Funcionario` construído, isto é, ainda se referirem para `null`.

5-) (Opcional) Crie um método para verificar se um determinado Funcionario se encontra ou não como funcionario desta empresa:

```
    boolean contem(Funcionario f) {
        // ...
    }
```

Você vai precisar fazer um for na sua array, e verificar se a referência passada como argumento se encontra dentro da array. Evite ao máximo usar números hard-

coded, isto é, use o `.length`.

6-) (Opcional) Caso a array já esteja cheia no momento de adicionar um outro funcionário, criar uma nova maior e copiar os valores. Isto é, fazer a realocação já que java não tem isso: uma array nasce e morre com o mesmo length.

Usando o this para passar argumento

Dentro de um método, você pode usar a palavra this para referenciar a si mesmo, e pode passar essa referência como argumento.

5.7 - Desafios

1-) No capítulo anterior, você deve ter reparado que a versão recursiva para o problema de Fibonacci é lenta porque toda hora estamos recalculando valores. Faça com que a versão recursiva seja tão boa quanto a versão iterativa.

5.8 - Testando o conhecimento

O objetivo dos exercícios a seguir é fixar os conceitos vistos até agora. Se você está com dificuldade em alguma parte desse capítulo, aproveite e treine tudo o que vimos até agora nos pequenos programas abaixo:

Programa 1

Classe: Casa
Atributos: cor, totalDePortas, portas[]
Método: void pinta(String s), int quantasPortasEstaoAbertas(), void adicionaPorta(Porta p), int totalDePortas()

Crie uma casa, pinte ela. Crie três portas e coloque-as na casa através do método adicionaPorta, abra e feche as mesmas como desejar. Utilize o método quantasPortasEstaoAbertas para imprimir o número de portas abertas e o método totalDePortas para imprimir o total de portas em sua casa.

capítulo 6

Modificadores de acesso e atributos de classe

"A marca do homem imaturo é que ele quer morrer nobremente por uma causa, enquanto a marca do homem maduro é querer viver modestamente por uma."

J. D. Salinger -

Ao término desse capítulo, você será capaz de:

- controlar o acesso aos seus métodos, atributos e construtores através dos modificadores private e public;
- escrever métodos de acesso a atributos do tipo getters e setters;
- escrever construtores para suas classes e
- utilizar variáveis e métodos estáticos.

6.1 - Controlando o acesso

Um dos problemas mais simples que temos no nosso sistema de contas é que a função saca permite sacar mesmo que o limite tenha sido atingido. A seguir você pode lembrar como está a classe Conta:

```
1.   class Conta {
2.    int numero;
3.    String dono;
4.    double saldo;
5.    double limite;
6.
7.    // ..
8.
9.    boolean saca(double quantidade) {
10.         this.saldo = this.saldo - quantidade;
11.   }
12.  }
```

A classe a seguir mostra como é possível ultrapassar o limite usando o método `saca`:

```
1.   class TestaContaEstouro1 {
2.    public static void main(String args[]) {
3.          Conta minhaConta = new Conta();
4.          minhaConta.saldo = 1000.0;
5.          minhaConta.limite = 1000.0;
6.          minhaConta.saca(50000); // saldo + limite é só 2000!!
7.    }
8.   }
```

Podemos incluir um `if` dentro do nosso método saca() para evitar a situação que resultaria em uma conta em estado inconsistente, com seu saldo abaixo do limite. Fizemos isso no capítulo de orientação a objetos básica.

Apesar de melhorar bastante, ainda temos um problema mais grave: ninguém garante que o usuário da classe vai sempre utilizar o método para alterar o saldo da conta. O código a seguir ultrapassa o limite diretamente:

```
1.    class TestaContaEstouro2 {
2.     public static void main(String args[]) {
3.           Conta minhaConta = new Conta();
4.           minhaConta.limite = 100;
5.           minhaConta.saldo = -200; //saldo está abaixo dos 100 de limite
6.     }
7.    }
```

Como evitar isso? Uma idéia simples seria testar se não estamos ultrapassando o limite toda vez que formos alterar o saldo:

```
class TestaContaEstouro3 {

      public static void main(String args[]) {
            // a Conta
            Conta minhaConta = new Conta();
            minhaConta.limite = 100;
            minhaConta.saldo = 100;

            // quero mudar o saldo para -200
            double novoSaldo = -200;

            // testa se o novoSaldo ultrapassa o limite da conta
            if (novoSaldo < -minhaConta.limite) { //
                  System.out.println("Não posso mudar para esse saldo");
            } else {
                  minhaConta.saldo = novoSaldo;
            }
      }
}
```

Esse código iria se repetir ao longo de toda nossa aplicação e, pior, alguém pode esquecer de fazer essa comparação em algum momento, deixando a conta na situação inconsistente. A melhor forma de resolver isso seria forçar quem usa a classe `Conta` a chamar o método `saca` e não permitir o acesso direto ao atributo. É o mesmo caso da validação de CPF.

PRIVATE

Para fazer isso no Java basta declarar que os atributos não podem ser acessados de fora da classe usando a palavra chave `private`:

```
class Conta {
      private double saldo;
      private double limite;
      // ...
}
```

MODIFICADOR DE ACESSO

`private` é um **modificador de acesso** (também chamado de **modificador de visibilidade**).

Marcando um atributo como privado, fechamos o acesso ao mesmo de todas as outras classes, fazendo com que o código não compile:

```
class TestaAcessoDireto {
      public static void main(String args[]) {
            Conta minhaConta = new Conta();
            // não compila! você não pode acessar o atributo privado de outra
classe
            minhaConta.saldo = 1000;
      }
}
```

Programando orientado a objetos é uma **prática obrigatória** proteger seus atributos como `private`. (discutiremos outros modificadores de acesso em outros capítulos).

Cada classe é responsável por controlar seus atributos, portanto ela deve julgar se aquele novo valor é válido ou não! Esta validação não deve ser controlada por quem está usando a classe e sim por ela mesma, centralizando essa responsabilidade.

A palavra chave `private` também pode ser usada para modificar o acesso a um método. Tal funcionalidade é normalmente usada quando existe um método apenas auxiliar a própria classe, e não queremos que outras pessoas o usem.

PUBLIC

Da mesma maneira que temos o `private`, temos o modificador `public`, que permite a todos acessar um determinado atributo ou método :

```
1.    class Conta {
2.       //...
3.
4.       public void saca(double quantidade) {
5.         if (quantidade > this.saldo + this.limite) { //posso sacar até
      saldo+limite
6.             System.out.println("Não posso sacar fora do limite!");
7.         } else {
8.             this.saldo = this.saldo - quantidade;
9.         }
10.      }
11.
12.   }
```

E quando não há modificador de acesso?

Até agora tínhamos declarado variáveis e métodos sem nenhum modificador como `private` e `public`. Quando isto acontece, o seu método ou atributo fica num estado de visibilidade intermediário entre o `private` e o `public`, que veremos mais pra frente, no capítulo de pacotes.

É muito comum, e faz todo sentido, que seus atributos sejam `private`, e quase todos seus métodos sejam `public` (não é uma regra!). Desta forma, toda conversa de uma classe com a outra é feita por troca de mensagens, isso é, acessando seus métodos, algo muito mais educado que mexer diretamente em um atributo que não é seu!

O conjunto de métodos públicos de uma classe é também chamado de **interface da classe**, pois esta é a maneira a qual você se comunica com essa classe.

Programando voltado para a interface e não para a implementação

É sempre bom programar pensando na interface da sua classe, como seus usuários estarão utilizando ela, e não somente como ela irá funcionar.

A implementação em si, o conteúdo dos métodos, não tem tanta importância para o usuário dessa classe uma vez que ele só precisa saber o que cada método pretende fazer, e não como ele faz pois isto pode mudar com o tempo.

Essa frase vem do livro Design Patterns, de Eric Gamma et al. Um livro cultuado no meio da orientação a objetos.

Repare que agora temos conhecimentos suficientes para estar resolvendo aquele problema da validação de CPF:

```
class Cliente {
      private String nome;
      private String endereco;
```

```
        private String cpf;
        private int idade;

        public void mudaCPF(String cpf) {
                validaCPF(cpf);
                this.cpf = cpf;
        }

        private void validaCPF(String cpf) {
                // série de regras aqui, falha caso nao seja válido
        }

        // ..
}
```

Agora, se alguém tentar criar um `Cliente` e não usar o `mudaCPF`, vai receber um erro de compilação, já que o atributo `CPF` é **privado**. E o dia que você não precisar validar quem tem mais de 60 anos? Seu método fica o seguinte:

```
public void mudaCPF(String cpf) {
        if (this.idade <= 60) {
                validaCPF(cpf);
        }
        this.cpf = cpf;
}
```

O controle sobre o `CPF` está centralizado: ninguém consegue acessá-lo sem passar por aí, a classe `Cliente` é a única responsável pelos seus próprios atributos!

6.2 - Getters e Setters

ENCAPSULAR

Um padrão que surgiu com o desenvolvimento de Java foi de **encapsular**, isto é, esconder todos os membros de uma classe como vimos acima. Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar dois métodos, um que retorna o valor e outro que muda o valor.

GETTERS
SETTERS

O padrão para esses métodos é de colocar a palavra `get` ou `set` antes do nome do atributo. Por exemplo, a nossa conta com saldo, limit e modelo fica assim:

```
1.    public class Conta {
2.
3.     private double saldo;
4.     private double limite;
5.     private Cliente dono;
6.
7.     public double getSaldo() {
8.           return this.saldo;
9.     }
10.
11.    public void setSaldo(double saldo) {
12.          this.saldo = saldo;
13.    }
14.
15.    public double getLimite() {
16.          return this.limite;
17.    }
18.
19.    public void setLimite(double limite) {
20.          this.limite = limite;
21.    }
22.
23.    public Cliente getDono() {
24.          return this.dono;
25.    }
26.
27.    public void setDono(Cliente dono) {
```

```
28.           this.dono = dono;
29.    }
30.   }
```

É uma má prática criar uma classe e logo em seguida criar getters e setters pros seus atributos. Você só deve criar um getter ou setter se tiver a real necessidade. Repare que nesse exemplo `setSaldo` não deveria ter sido criado, já que queremos que todos usem `retira`() e `saca`().

Outro detalhe importante, um método `getX` não necessariamente retorna o valor de um atributo que chama `X` do objeto em questão. Isso é interessante para o encapsulamento. Imagine a situação: queremos que o banco sempre mostre como saldo o valor do limite somado ao `saldo` (uma prática comum dos bancos que costuma iludir seus clientes). Poderíamos sempre chamar c.`getLimite`() + c.`getSaldo`(), mas isso poderia gerar uma situação de "replace all" quando precisassemos mudar como o saldo é mostrado. Podemos encapsular isso em um método, e porque não dentro do próprio `getSaldo`? Repare:

```
31.  public class Conta {
32.
33.   private double saldo;
34.   private double limite;
35.   private Cliente dono;
36.
37.   private double getSaldo() {
38.           return this.saldo + this.limite;
39.   }
40.
41.   // deposita() e saca()
42.
43.   public Cliente getDono() {
44.           return this.dono;
45.   }
46.
47.   public void setDono(Cliente dono) {
48.           this.dono = dono;
49.   }
50.  }
```

O código acima nem possibilita a chamada do método `getLimite`(), ele não existe. E nem deve existir enquanto não houver essa necessidade. O método `getSaldo` não devolve simplesmente o saldo... e sim o que queremos que seja mostrado como se fosse o `saldo`. Utilizar getter e setters não só ajudam você a proteger seus atributos, como também possibilita ter de mudar algo em um só lugar... chamamos isso de encapsulamento, pois esconde a maneira como seus objetos guardam seus dados. É uma prática muito importante.

Nossa classe está agora totalmente pronta? Isto é, existe a chance dela ficar com menos dinheiro do que o limite? Pode parecer que não, mas e se depositarmos um valor negativo na conta? Ficaríamos com menos dinheiro que o permitido, já que não esperávamos por isso. Para nos proteger disso basta mudarmos o método `deposita`() para que ele verifique se o valor é necessariamente positivo. Depois disso precisaríamos mudar mais algum outro código? A resposta é não, graças ao encapsulamento dos nossos dados.

6.3 - Exercícios

1-) Uma vez que os **getters** e **setters** são uma base importante para tudo que será feito em Java, crie os métodos get e set para todos os seus atributos da classe `Conta`. Aproveite e mantenha essa nova classe em um diretório novo e comece a

trabalhar agora nesse diretório.

Assim você não terá a necessidade de alterar seus programas que estão funcionando e a partir de agora irá utilizar o jeito correto de trabalhar com atributos.

6.4 - Construtores

CONSTRUTOR

Quando usamos a palavra chave `new`, estamos construindo um objeto. Sempre quando o new é chamado, executa o **construtor da classe**. O construtor da classe é um bloco declarado com o **mesmo nome** que a classe:

```
1.    class Conta {
2.     int numero;
3.     String dono;
4.     double saldo;
5.     double limite;
6.
7.     // construtor
8.     Conta() {
9.           System.out.println("Construindo uma conta.");
10.    }
11.
12.    // ..
13.   }
```

Então, quando fizermos:

```
Conta c = new Conta();
```

A mensagem "construindo uma conta" aparecerá.

O construtor default

Até agora, as nossas classes não possuíam nenhum construtor. Então como é que era possível dar `new`, se todo `new` chama um construtor **obrigatoriamente**?

Quando você não declara nenhum construtor na sua classe, o Java cria um para você. Esse construtor é o **construtor default**, ele não recebe nenhum argumento e o corpo dele é vazio.

A partir do momento que você declara um construtor, o construtor default não é mais fornecido.

O interessante é que um construtor pode receber um argumento, podendo assim inicializar algum tipo de informação:

```
1.    class Conta {
2.     int numero;
3.     String dono;
4.     double saldo;
5.     double limite;
6.
7.     // construtor
8.     Conta(String dono) {
9.           this.dono = dono;
10.    }
11.
12.    // ..
13.   }
```

Esse construtor recebe o dono da conta. Assim, quando criarmos uma conta, ela já terá um determinado dono.

```
Conta c = new Conta("Duke");
System.out.println(c.dono);
```

6.5 - O Motivo

Tudo estava funcionando até agora. Para que utilizamos um construtor?

A idéia é bem simples. Se toda conta precisa de um dono, como obrigar todos os objetos que forem criados a ter um valor desse tipo? Basta criar um único construtor que recebe essa String!

O construtor se resume a isso! Dar possibilidades ou obrigar o usuário de uma classe de passar argumentos para o objeto durante o processo de criação do mesmo.

Por exemplo, não podemos abrir um arquivo para leitura sem dizer qual é o nome do arquivo que desejamos ler! Portanto nada mais natural que passar uma `String` representando o nome de um arquivo na hora de criar um objeto do tipo de leitura de arquivo, e que isso seja obrigatório.

Você pode ter mais de um construtor na sua classe, e no momento do `new`, o construtor apropriado será escolhido.

Construtor: um método especial?

Um construtor não é um método. Algumas pessoas o chamam de um método especial, mas definitivamente não é, já que não possui retorno e só é chamado durante a construção do objeto.

Chamando outro construtor

Um construtor só pode rodar durante a construção do objeto, isto é, você nunca conseguirá chamar o construtor em um objeto já construído. Porém, durante a construção de um objeto, você pode fazer com que um construtor chame outro, para não ter de ficar copiando e colando:

```
class Conta {
	int numero;
	String dono;
	double saldo;
	double limite;

	// construtor
	Conta(String dono) {
		// faz mais uma série de inicializações e configurações
		this.dono = dono;
	}

	public Conta (int numero, String dono) {
		this(dono); // chama o construtor que foi declarado acima
		this.dono = dono;
	}

	// ..
}
```

Existe um outro motivo, o outro lado dos construtores: preguiça. Às vezes criamos um construtor que recebe diversos argumentos para não obrigar o usuário de uma classe a chamar diversos métodos do tipo '`set`'.

6.6 - Atributos de classe

Nosso banco também quer controlar a quantidade de contas existentes no sistema. Como poderíamos fazer isto? A idéia mais simples:

```
Conta c = new Conta();
totalDeContas = totalDeContas + 1;
```

Aqui voltamos em um problema parecido com o da validação de CPF. Estamos espalhando um código por toda aplicação, e quem garante que vamos conseguir lembrar de incrementar a variável `totalDeContas` toda vez?

Tentamos então, passar para a seguinte proposta:

```
class Conta {
	private int totalDeContas;
	//...

	Conta() {
		this.totalDeContas = this.totalDeContas + 1;
	}
}
```

Quando criarmos duas contas, qual será o valor do `totalDeContas` de cada uma delas? Vai ser 1. Pois cada uma tem essa variável. **O atributo é de cada objeto**.

STATIC

Seria interessante então, que essa variável fosse **única**, compartilhada por todos os objetos dessa classe. Dessa maneira, quando mudasse através de um objeto, o outro enxergaria o mesmo valor. Para fazer isso em java, declaramos a variável como `static`.

```
private static int totalDeContas;
```

Quando declaramos um atributo como `static`, ele passa a não ser mais um atributo de cada objeto, e sim um **atributo da classe**, a informação fica guardada pela classe, não é mais individual para cada objeto.

Para acessarmos um atributo estático, não usamos a palavra chave `this`, e sim o nome da classe:

```
class Conta {
	private static int totalDeContas;
	//...

	Conta() {
		Conta.totalDeContas = Conta.totalDeContas + 1;
	}
}
```

Já que o atributo é privado, como podemos acessar essa informação a partir de outra classe? Precisamos de um getter para ele!

```
class Conta {
	private static int totalDeContas;
	//...

	Conta() {
		Conta.totalDeContas = Conta.totalDeContas + 1;
	}

	public int getTotalDeContas() [
		return Conta.totalDeContas;
	}
```

```
}
```

Como fazemos então para saber quantas contas foram criadas?

```
Conta c = new Conta();
int total = c.getTotalDeContas();
```

Precisamos criar uma cconta antes de chamar o método! Isso não é legal, pois gostaria de saber quantas contas existem sem precisar ter acesso a um objeto conta. A idéia aqui é a mesma, transformar esse método que todo objeto conta tem, para ser um método de toda a classe. Usamos a palavra static de novo, mudando o método anterior.

```
public static int getTotalDeContas() [
       return Conta.totalDeContas;
}
```

Para acessar esse novo método:

```
int total = Conta.getTotalDeContas();
```

Repare que estamos chamando um método não com uma referência para um carro, e sim usando o nome da classe.

Métodos e atributos estáticos

Métodos e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma classe.

6.7 - Um pouco mais...

1-) Em algumas empresas, o UML é amplamente utilizado. Às vezes, o programador recebe o UML já pronto, completo, e só deve preencher a implementação, devendo seguir a risca o UML. O que você acha dessa prática? Vantagens e desvantagens.

2-) Se uma classe só tem atributos e métodos estáticos, que conclusões podemos tirar? O que lhe parece um método estático?

3-) O padrão dos métodos `get` e `set` não vale para as variáveis de tipo `boolean`. Esses atributos são acessados via `is` e `set`. Por exemplo, para verificar se um carro está ligado seriam criados os métodos `isLigado` e `setLigado`.

6.8 - Exercícios

1-) Adicione o modificador de visibilidade (`private` ou `public`) para cada atributo e método da classe `Funcionario` e `Empresa`. Tente criar um `Funcionario` no `main` e modificar ou ler um de seus atributos privados. O que acontece?

2-) Crie os getters e setters da sua classe `Funcionario` e `Empresa`. Lembre-se de que não necessariamente todos os atributos devem ter getters e setters.

Por exemplo, na classe `Empresa`, seria interessante ter um setter e getter para a sua array de funcionarios? Não seria mais interessante ter um método como este: ?

```
class Empresa {
       // ...
```

```
        Funcionario getFuncionario(int posicao) {
              return this.funcionarios[posicao];
        }
}
```

3-) Adicione um atributo na classe `Funcionario` de tipo `int` que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo `Funcionario`. O primeiro `Funcionario` instanciado tem identificador 1, o segundo 2, e assim por diante. Você deve utilizar os recursos aprendidos aqui para resolver esse problema.

Crie um getter para o identificador. Devemos ter um setter?

4-) (opcional) Na classe `Empresa`, em vez de cirar uma array de tamanho fixo, receba como parâmetro no construtor o tamanho da array de `Funcionario`

Agora com esse construtor, o que acontece se tentarmos dar `new Empresa()` sem passar argumento algum? Porque?

5-) (opcional) Como garantir que datas como 31/2/2005 não sejam aceitas pela sua classe `Data`?

6-) (opcional) Crie a classe `PessoaFisica`. Queremos ter a garantia que pessoa fisica alguma tenha CPF invalido, nem seja criada `PessoaFisica` sem cpf inicial. (você não precisa escrever o algoritmo de validação de cpf, basta passar o cpf por um método `valida(String x)`....)

7-) (opcional) Porque esse código não compila?

```
class Teste {
      int x = 37;
      public static void main(String [] args) {
            System.out.println(x);
      }
}
```

6.9 - Desafios

1-) Imagine que tenho uma classe `FabricaDeCarro` e quero garantir que só existe um objeto desse tipo em toda a memória. Não existe uma palavra chave especial para isto em java, então teremos de fazer nossa classe de tal maneira que ela respeite essa nossa necessidade. Como fazer isso?

capítulo 7

Orientação a Objetos – herança, reescrita e polimorfismo

"O homem absurdo é aquele que nunca muda."
Georges Clemenceau -

Ao término desse capítulo, você será capaz de:
- dizer o que é herança e quando utilizá-la;
- reutilizar código escrito anteriormente;
- criar classes filhar e reescrever métodos;
- usar todo o poder que o polimorfismo da.

7.1 - Repetindo código?

Como toda empresa, nosso Banco possui funcionários. Vamos modelar a classe Funcionario:

```
class Funcionario {

       String nome;
       String cpf;
       double salario;

       // métodos devem vir aqui
}
```

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes. Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco:

```
class Gerente {

       String nome;
       String cpf;
       double salario;

       int senha;

       public boolean autentica(int senha) {
              if (this.senha == senha) {
                     System.out.println("Acesso Permitido!");
                     return true;
              } else {
                     System.out.println("Acesso Negado!");
                     return false;
              }
       }

       // outros métodos
}
```

Precisamos mesmo de outra classe?

Poderíamos ter deixado a classe `Funcionario` mais genérica, mantendo nela senha de acesso. Caso o funcionário não fosse um gerente, deixaríamos este atributo vazio.

Essa é uma possibilidade. Mas e em relação aos métodos? A classe `Gerente` tem o método autentica, que não faz sentido ser acionado em um funcionário que não é gerente.

Se tivéssemos um outro tipo de funcionário, que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe, e copiar o código novamente!

Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaríamos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizar as informações principais do funcionário em um único lugar!

HERANÇA

Existe uma maneira, em Java, de relacionarmos uma classe de tal maneira que uma delas **herda** tudo que a outra tem. Isto é uma relação de classe mãe e classe filha. No nosso caso, gostaríamos de fazer com que o `Gerente` tivesse tudo que um `Funcionario` tem, gostaríamos que ela fosse uma **extensão** de `Funcionario`. Fazemos isto através da palavra chave `extends`.

EXTENDS

```
class Gerente extends Funcionario {

	int senha;

	public boolean autentica(int senha) {
		if (this.senha == senha) {
			System.out.println("Acesso Permitido!");
			return true;
		} else {
			System.out.println("Acesso Negado!");
			return false;
		}
	}

}
```

Todo momento que criarmos um objeto do tipo `Gerente`, este objeto possuirá também os atributos definidos na classe `Funcionario`, pois agora um `Gerente` **é um** `Funcionario`:

```
Gerente gerente = new Gerente();
gerente.nome = "João da Silva";
gerente.senha = "4231";
```

Dizemos que a classe `Gerente` **herda** todos os atributos e métodos da classe mãe, no nosso caso, a `Funcionario`. Para ser mais preciso, ela também herda os atributos e métodos privados, porém não consegue acessa-los diretamente.

SUPER E SUB CLASSES

Super e Sub classe

A nomenclatura mais encontrada é que Funcionario é a **superclasse** de Gerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente é um Funcionário.

PROTECTED

E se precisamos acessar os atributos que herdamos? Não gostaríamos de deixar os atributos de `Funcionario` `public`, pois dessa maneira qualquer um

poderia alterar os atributos dos objetos deste tipo. Existe um outro modificador de acesso, o `protected`, que fica entre o `private` e o `public`. Um atributo `protected` só pode ser acessado (visível) pela própria classe ou suas subclasses.

```
class Funcionario {

       protected String nome;
       protected String cpf;
       protected double salario;

       // métodos devem vir aqui
}
```

Sempre usar protected?

Então porque usar `private`? Depois de um tempo programando orientado a objetos, você vai começar a sentir que nem sempre é uma boa idéia deixar que a classe filha acesse os atributos da classe mãe, pois isto quebra um pouco a idéia de que só aquela classe deveria manipular seus atributos. Essa é uma discussão um pouco mais avançada.

Além disso, não só as subclasses podem acessar os atributos `protected`, como outras classes, que veremos mais a frente (mesmo pacote).

7.2 - Reescrita de método

Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

Vamos ver como fica a classe `Funcionario`:

```
class Funcionario {

       protected String nome;
       protected String cpf;
       protected double salario;

       public double getBonificacao() {
              return this.salario * 0.10;
       }
       // métodos
}
```

Se deixarmos a classe `Gerente` como ela está, ela vai herdar o método `getBonificacao`.

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

REESCRITA

O resultado aqui será 500. Não queremos essa resposta, não queremos este método que foi escrito na classe mãe, eu quero **reescrever** (**sobrescrever**, **override**) este método:

```
class Gerente extends Funcionario {

       int senha;

       public double getBonificacao() {
              return this.salario * 0.15;
       }

       // ...
```

```
}
```

Agora sim o método está correto para o `Gerente`. Refaça o teste e veja que, agora, o valor impresso é o correto (750):

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

7.3 - Polimorfismo

O que guarda uma variável do tipo `Funcionario`? Uma referência para um `Funcionario`.

Na herança, vimos que `Gerente` é um `Funcionario`, pois é uma extensão deste. Eu posso me referenciar a um `Gerente` como sendo um `Funcionario`. Se alguém precisa falar com um `Funcionario` do banco, pode falar com um `Gerente`! Por que? Pois `Gerente` **é um** `Funcionario`. Essa é a semântica da herança.

```
Funcionario funcionario = new Gerente();
funcionario.salario = 5000.0;
```

POLIMORFISMO

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo nao quer dizer que o objeto fica se transformando, muito pelo contrario, um objeto nasce de um tipo e morre daquele tipo, o que muda e a maneira como nos referenciamos a ele).

Até aqui tudo bem, mas e se eu tentar:

```
funcionario.getBonificacao();
```

Qual é o retorno desse método? 500 ou 750? No Java, a chamada de método sempre vai ser **decidida em tempo de execução**. O Java vai procurar o objeto na memória e ai sim decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não a que estamos usando para referencia-lo. Apesar de estarmos nos referenciando a esse `Gerente` como sendo um `Funcionario`, o método executado é o do `Gerente`. O retorno é 750.

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Porque faria isso? Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo `Funcionario`:

```
class ControleDeBonificacoes {
	private double totalDeBonificacoes = 0;

	public void bonifica(Funcionario funcionario) {
		this.totalDeBonificacoes += funcionario.getBonificacao();
	}

	public double getTotalDeBonificacoes() {
		return this.totalDeBonificacoes;
	}
}
```

E, em algum lugar da minha aplicação (ou no main se for apenas para testes):

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();
Gerente funcionario1 = new Gerente();
funcionario1.setSalario(5000.0);
controle.bonifica(funcionario1);
```

```
Funcionario funcionario2 = new Funcionario();
funcionario2.setSalario(1000.0);
controle.bonifica(funcionario2);

System.out.println(controle.getTotalDeBonificacoes());
```

Repare que conseguimos passar um `Gerente` para um método que recebe um `Funcionario` como argumento.

Qual será o valor resultante?

O dia que criarmos uma classe `Secretaria`, por exemplo, que é filha de `Funcionario`, precisaremos mudar a classe de `ControleDeBonificacoes`? Não. Basta a classe `Secretario` reescrever os métodos que lhe parecer necessário. É exatamente esse o poder do polimorfismo juntamente com a herança e reescrita de método: diminuir acoplamento entre as classes, para que evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou `ControleDeBonificacoes` pode nunca ter imaginado a criação da classe `Secretario ou Engenheiro. Isto tras um reaproveitamente enorme de código.`

7.4 - Um exemplo mais completo

Imagine que vamos modelar um sistema para a faculdade, que controle as despesas com funcionários e professores. Nosso funcionário fica assim:

```
class EmpregadoDaFaculdade {
	private String nome;
	private double salario;

	double getGastos() {
		return this.salario;
	}

	String getInfo() {
		return "nome: " + this.nome + " com salário " + this.salario;
	}

	// métodos de get, set e outros
}
```

O gasto que temos com o professor não é apenas seu salário. Temos de somar um bônus de 10 reais por hora/aula. O que fazemos então? Reescrevemos o método. Assim como o `getGastos` é diferente, o `getInfo` também será, pois temos de mostrar as horas aula também.

```
class ProfessorDaFaculdade extends EmpregadoDaFaculdade {
	private int horasDeAula;

	double getGastos() {
		return this.getSalario() + this.horasDeAula * 10;
	}

	String getInfo() {
		String informacaoBasica = super.getInfo();
		String informacao = informacaoBasica + " horas de aula: " +
					this.horasDeAula;
		return informacao;
	}

	// métodos de get, set e outros
}
```

A novidade aqui é a palavra chave super. Apesar do método ter sido reescrito, gostaríamos de acessar o método da classe mãe, para não ter de copiar e colocar o conteúdo desse método e depois concatenar com a informação das horas de aula.

Como tiramos proveito do polimorfismo? Imagine que temos uma classe de relatório:

```
class GeradorDeRelatorio {
       public void adiciona(FuncionarioDaFaculdade f) {
              System.out.println(f.getInfo());
              System.out.println(f.getGatos());
       }
}
```

Podemos passar para nossa classe qualquer `FuncionarioDaFaculdade`! Vai funcionar tanto para professor, quanto para funcionário comum.

Um certo dia, muito depois de terminar essa classe de relatório, resolvemos aumentar nosso sistema, e colocar uma classe nova, que representa o `Reitor`. Como ele também é um `FuncionarioDaFaculdade`, será que vamos precisar alterar alguma coisa na nossa classe de `Relatorio`? Não. essa é a idéia. Quem programou a classe `GeradorDeRelatorio` nunca imaginou que existiria uma classe `Reitor`, e mesmo assim o sistema funciona.

```
class Reitor extends ProfessorDaFaculdade {
       // informações extras

       String getInfo() {
              return super.getInfo() + " e ele é um reitor";
       }

       // não sobreescrevemos o getGastos!!!
}
```

7.5 - Um pouco mais...

1-) Se não houvesse herança em Java, como você poderia reaproveitar o código de outra classe?

COMPOSIÇÃO

2-) Uma discussão muito atual é sobre o abuso no uso da herança. Algumas pessoas usam herança apenas para reaproveitar o código, quando poderia ter feito uma **composição**. Procure sobre herança versus composição.

3-) Mesmo depois de reescrever um método da classe mãe, a classe filha ainda pode acessar o método antigo. Isto é feito através da palavra chave `super.método()`. Algo parecido ocorre entre os construtores das classes, o que?

7.6 - Exercícios

1-) Vamos criar uma classe `Conta`, que possua um saldo, e os métodos para pegar saldo, depositar, e retirar. O esqueleto é o seguinte:

```
class Conta {
       private double saldo;

       void deposita(double x) {
              //...
       }
       void retira(double x) {
              // ..
       }
       double getSaldo() {
```

```
            //..
        }
}
```

Preencha o esqueleto conforme exemplos vistos no decorrer dos outros capítulos. Se achar interessante faça com que o método `retira()` retorne um `boolean` indicando o sucesso da operação.

2-) Adicione um método na classe `Conta`, que atualiza essa conta de acordo com a taxa selic fornecida.

```
    void atualiza(double taxaSelic) {
            this.saldo = this.saldo * (1 + taxaSelic);
    }
```

3-) Crie duas subclasses da classe `Conta`: `ContaCorrente` e `ContaPoupanca`. Ambas terão o método atualiza reescrito: A `ContaCorrente` deve atualizar-se com o dobro da taxaSelic, porém subtraindo 15 reais de taxa bancária. A `ContaPoupanca` deve atualizar-se com 75% da taxaSelic.

```
class ContaCorrente extends Conta {
        void atualiza(double taxaSelic) {
                // fazer conforme enunciado
        }
}

class ContaPoupanca extends Conta {
        void atualiza(double taxaSelic) {
                // fazer conforme enunciado
        }
}
```

Repare que para acessar o atributo saldo herdado da classe `Conta`, você vai precisar trocar o modificador de visibilidade de saldo para protected. Você pode utilizar os métodos retira e deposita se preferir continuar com o `private` (recomendado!), ou então criar um método `setSaldo`, mas `protected`, para não deixar outras pessoas alterarem o saldo dessa maneira..

4-) Crie uma classe com método `main` e instacie essas classes, atualize-as e veja o resultado. Algo como:

```
Conta c = new Conta();
ContaCorrente cc = new ContaCorrente();
ContaPoupanca cp = new ContaPoupanca();

c.deposita(1000);
cc.deposita(1000);
cp.deposita(1000);

c.atualiza(0.01);
cc.atualiza(0.01);
cp.atualiza(0.01);
```

Agora imprima o saldo (getSaldo()) de cada uma das contas, o que acontece?

5-) O que você acha de rodar o código anterior da seguinte maneira:

```
Conta c = new Conta();
Conta cc = new ContaCorrente();
Conta cp = new ContaPoupanca();
```

Compila? Roda? O que muda? Qual é a utilidade disso?

6-) (opcional) Vamos criar uma classe que seja responsável por fazer a atualização de todas as contas bancárias, e gerar um relatório com o saldo anterior e saldo novo de cada uma das contas.

```
class AtualizadorDeContas {
	private double saldoTotal = 0;
	private double selic;

	AtualizadorDeContas(double selic) {
		this.selic = selic;
	}

	void roda(Conta c) {
		// aqui voce imprime o saldo anterior, atualiza a conta,
		// e depois imprime o saldo final
		// lembrando de somar o saldo final ao atributo saldoTotal
	}
	// outros métodos
}
```

7-) (opcional) No método `main`, vamos criar algumas contas e passa-la

```
AtualizadorDeContas adc = new AtualizadorDeContas(0.08);
adc.roda(c);
adc.roda(cc);
adc.roda(cp);
System.out.println(adc.getSaldoTotal());
```

8-) (Opcional) Crie uma classe `Banco` que possui uma array de `Conta`. Repare que numa array de `Conta` você pode colocar tanto `ContaCorrente` quanto `ContaPoupanca`. Crie um metodo `void adiciona(Conta c)`, um método `Conta pega(int x)` e outro `int pegaTotalDeContas()`, muito similar a relação anterior de Empresa-Funcionario.

Faça com que seu método `main` crie diversas contas, insira-as no `Banco`, e depois com um `for` percorra todas as contas do `Banco` para passá-las como argumento para o `AtualizadorDeContas`.

9-) (Opcional) Use a palavra chave super nos métodos atualiza reescritos, para não ter de refazer o trabalho.

10-) (Opcional) Se você precisasse criar uma classe `ContaInvestimento`, e seu método atualiza fosse complicadíssimo, você precisaria alterar as classes `Banco` e `AtualizadorDeContas`?

capítulo 8

Orientação a Objetos – Classes Abstratas

"Dá-se importância aos antepassados quando já não temos nenhum."
François Chateaubriand -

Ao término desse capítulo você será capaz de utilizar classes abstratas quando necessário.

8.1 - Repetindo mais código?

Neste capítulo aconselhamos que você passe a usar o Eclipse. Você já tem conhecimento suficiente dos erros de compilação do javac, e agora pode aprender as facildiades que o Eclipse te traz ao ajudar você no código com os chamados quick fixes e quick assists.

Vamos recordar em como pode estar nossa classe `Funcionario`:

```
class Funcionario {

	protected String nome;
	protected String cpf;
	protected double salario;

	public double getBonificacao() {
		return this.salario * 1.2;
	}


	// outros métodos aqui
}
```

Considere agora o nosso `ControleDeBonificacao`:

```
class ControleDeBonificacoes {
	private double totalDeBonificacoes = 0;

	public void registra(Funcionario f) {
		System.out.println("Adicionando bonificacao do funcionario: " + f);
		this.totalDeBonificacoes += funcionario.getBonificacao();
	}

	public double getTotalDeBonificacoes() {
		return this.totalDeBonificacoes;
	}
}
```

Nosso método `registra` recebe qualquer referencia do tipo `Funcionario`, isto é, pode ser objetos do tipo `Funcionario` e qualquer de seus subtipos: `Gerente`, `Diretor` e eventualmente alguma nova subclasse que venha ser escrita, sem prévio conhecimento do autor da `ControleDeBonificacao`.

Estamos utilizando aqui a classe `Funcionario` para o polimorfismo: se não

fosse ela teríamos um grande prejuízo: precisaríamos criar um método `bonifica` para receber cada um dos tipos de `Funcionario`, um para `Gerente`, um para `Diretor`, etc. Repare que perder esse poder é muito pior que a pequena vantagem que a herança traz em herdar código.

Porém em alguns sistemas, como é o nosso caso, usamos uma classe apenas com esses intuitos: de economizar um pouco código e ganhar polimorfismo para criar métodos mais genéricos e que se encaixem a diversos objetos. Faz sentido ter um objeto do tipo Funcionario?

Essa pergunta é diferente de saber se faz sentido ter uma referência do tipo `Funcionario`: esse caso faz sim e é muito útil. Referenciando `Funcionario` temos o polimorfismo de referência, já que podemos receber qualquer coisa que seja um `Funcionario`. Porém dar `new` em `Funcionario` pode não fazer, isso é, não queremos receber um objeto do tipo Funcionario, queremos que aquela referência seja ou um `Gerente`, ou um `Diretor`, etc. Algo mais **concreto** que um `Funcionario`.

```
ControleDeBonificacoes cdb = new ControleDeBonificacoes();
Funcionario f = new Funcionario();
cdb.adiciona(f); // faz sentido?
```

Um outro caso em que não faz sentido ter um objeto daquele tipo, apesar da classe existir: imagine a classe `Pessoa` e duas filhas, `PessoaFisica` e `PessoaJuridica`. Quando puxamos um relatório de nossos clientes (uma array de Pessoa por exemplo), queremos que cada um deles seja ou uma `PessoaFisica`, ou uma `PessoaJuridica`. A classe `Pessoa` nesse caso estaria sendo usada apenas para ganhar o polimorfismo e herdar algumas coisas: não faz sentido permitir instanciá-la.

Para resolver esses problemas temos as classes abstratas.

8.2 - Classe abstrata

O que exatamente vem a ser a nossa classe `Funcionario`? Nossa empresa tem apenas Diretores, Gerentes, Secretarios, etc. Ela é uma classe que apenas idealiza um tipo, define apenas um rascunho.

Para o nosso sistema é inadmissível um objeto ser apenas do tipo `Funcionario` (pode existir um sistema em que faça sentido ter objetos do tipo `Funcionario` ou apenas `Pessoa`, mas no nosso caso não).

CLASSE ABSTRATA

ABSTRACT

Usamos a palavra chave `abstract` para impedir que ela possa ser instanciada. Esse é o efeito diretor de se usar o modificador abstract na declaração de uma classe:

```
abstract class Funcionario {

	protected double salario;

	public double getBonificacao() {
		return this.salario * 1.2;
	}

	// outros atributos e métodos comuns a todos Funcionarios

}
```

E no meio de um código:

```
Funcionario f = new Funcionario() // não compila!!!
```

O código acima não compila. O problema é instanciar a classe, criar referência você pode (e deve, pois é útil). Se ela não pode ser instanciada, para que serve? Somente para o polimorfismo e herança dos atributos e métodos.

Vamos então herdar dessa classe, reescrevendo o método `getBonificacao`:

```
class Gerente extends Funcionario {

      public String getBonificacao() {
            return this.salario * 1.4 + 1000;
      }
}
```

Mas qual é a real vantagem de uma classe abstrata? Poderíamos ter feito isto com uma herança comum. Por enquanto, a única diferença é que não podemos instanciar um objeto do tipo `Funcionario`, que já é de grande valia, dando mais consistência ao sistema.

8.3 - Métodos abstratos

Se não tivéssemos reescrito o método `getBonificacao`, esse método seria herdado da classe mãe, fazendo com que ele devolvesse o salário mais 20%. Cada funcionário em nosso sistema tem uma regra totalmente diferente para ser bonificado.

Será então que faz algum sentido ter esse método na classe `Funcionario`? Será que existe uma bonificação padrão para todo tipo de `Funcionario`?Parece não... cada classe filha terá um método diferente de bonificação pois de acordo com nosso sistema não existe uma regra geral: queremos que cada pessoa que escreve a classe de um `Funcionario` diferente (subclasses de `Funcionario`) reescreva o método `getBonificacao` de acordo com as suas regras.

Poderíamos então jogar fora esse método da classe `Funcionario`? O problema é que se ele não existisse, não poderíamos chamar o método apenas com uma referência a um `Funcionario`, pois ninguém garante que essa referência aponta para um objeto que possui esse método.

MÉTODO ABSTRATO

Existe um recurso em Java que, em uma classe abstrata, podemos escrever que determinado método será **sempre** escrito pelas classes filhas. Isto é, um **método abstrato**.

Ele indica que todas as classes filhas (concretas, isso é, que não forem abstratas) devem reescrever esse método, ou não compilarão. É como se você herdasse a responsabilidade de ter aquele método.

Como declarar um método abstrato

Às vezes não fica claro como declarar um método abstrato.
Basta escrever a palavra chave abstract na assinatura do mesmo e colocar um ponto e vírgula em vez de abre e fecha chaves!

```
abstract class Funcionario {

      abstract double getBonificacao();
```

```
    // outros atributos e métodos

}
```

Repare que não colocamos o corpo do método, e usamos a palavra chave `abstract` para definir o mesmo. Porque não colocar corpo algum? Porque esse método nunca vai ser chamado, sempre quando alguém chamar o método `getBonificacao`, vai cair em uma das suas filhas, que realmente escreveram o método.

Qualquer classe que estender a classe `Funcionario` será obrigada a reescrever este método, tornando-o "concreto". Se não reescreverem esse método, um erro de compilação ocorrerá.

O método do `ControleDeBonificacao` estava assim:

```
public void registra(Funcionario f) {
      System.out.println("Adicionando bonificacao do funcionario: " + f);
      this.totalDeBonificacoes += funcionario.getBonificacao();
}
```

Como posso acessar o método `getBonificacao` se ele não existe na classe `Funcionario`?

Já que o método é abstrato, **com certeza** suas subclasses têm esse método, o que garante que essa invocação de método não vai falhar. Basta pensar que uma referência do tipo `Funcionario` nunca aponta para um objeto que não tem o método `getBonificacao`, pois não é possível instanciar uma classe abstrata, apenas as concretas. Um método abstrato obriga a classe em que ele se encontra ser abstrata, o que garante a coerência do código acima compilar.

8.4 - Um outro exemplo

Nosso banco deseja todo dia de manhã atualizar as contas bancárias de todas as pessoas. Temos dois tipos de conta, a `ContaCorrente` e a `ContaPoupanca`. A `ContaPoupanca` atualiza todo dia uma pequena porcentagem, já a `ContaCorrente` só precisa atualizar-se com um fator de correção mensal.

```
1.  class Conta {
2.     private double saldo = 0.0;
3.
4.     public void retira(double valor) {
5.           this.saldo -= valor;
6.     }
7.
8.     public void deposita(double valor) {
9.           this.saldo += valor;
10.    }
11.
12.    public double getSaldo() {
13.          return this.saldo();
14.    }
15. }
```

```
1.  class ContaCorrente extends Conta {
2.     private double limiteDoChequeEspecial = 1000.0;
3.     private double gastosNoChequeEspecial = 100.0;
4.
5.     public void atualiza() {
6.           super.retira(this.gastosNoChequeEspecial * 0.08);
7.     }
8.  }
```

```
1.  class ContaPoupanca extends Conta {
2.     private double correcaoMensal;
3.
4.     public void atualiza() {
5.           super.deposita(this.saldo * this.correcaoMensal);
6.     }
7.  }
```

O que não está legal aqui? Por enquanto usamos herança para herdar um pouco de código, e assim não ter de reescreve-lo. Mas já frisamos que essa não é a grande vantagem de se usar herança, a idéia é utilizar o polimorfismo adquirido. Podemos nos referenciar a uma `ContaCorrente` e `ContaPoupanca` como sendo uma `Conta`:

```
class AtualizadorDeSaldos {
      private Conta[] contas;

      public void setContas(Conta[] contas) {
            this.contas = contas;
      }

      public void atualizaSaldos() {
            for (Conta conta : this.contas) {
                  conta.atualiza(); // não compila!!!
            }
      }
}
```

Este código acima não compila! Se tenho uma referência para uma `Conta`, quem garante que o objeto referenciado tem o método atualiza? Ninguém. Podemos então coloca-lo na classe `Conta`:

```
class Conta {
      protected double saldo;

      public void retira(double valor) {
            this.saldo -= valor;
      }

      public void deposita(double valor) {
            this.saldo -= valor;
      }

      public double getSaldo() {
            return this.saldo();
      }

      public void atualiza() {
            // não faz nada, serve só para o polimorfismo
      }
}
```

O que ainda não está legal? Cada tipo de `Conta`, isto é, cada subclasse de `Conta` sabe como se atualizar. Só que quando herdamos de `Conta` nós já herdamos o método atualiza, o que não nos obriga a reescreve-lo. Além disso, no nosso sistema não faz sentido existir um objeto que é realmente da classe `Conta`, essa classe é só um conceito, uma idéia, ela é abstrata! Assim como seu método atualiza, o qual queremos forçar que as subclasse reescrevam.

```
abstract class Conta {
      protected double saldo;

      public void retira(double valor) {
            this.saldo -= valor;
      }

      public void deposita(double valor) {
```

```
            this.saldo += valor;
      }

      public double getSaldo() {
            return this.saldo;
      }

      public abstract void atualiza();
}
```

Podemos então testar esses conceitos criando 2 Contas (uma de cada tipo) e chamando o método atualiza de cada uma delas:

```
public class TesteClassesAbstratas {
      public static void main (String args[]) {

            //criamos as contas
            Conta[] contas = new Conta[2];
            contas[0] = new ContaPoupanca();
            contas[1] = new ContaCorrente();

            //iteramos e chamamos atualiza
            for (Conta conta : contas) {
                  conta.atualiza();
            }
      }
}
```

java.io

Classes abstratas não possuem nenhum segredo no aprendizado, mas quem está aprendendo orientação a objetos pode ter uma enorme dificuldade para saber quando utilizá-las, o que é muito normal.

Estudaremos o pacote java.io, que usa bastante classes abstratas, sendo um exemplo real de uso desse recurso, que vai melhorar o entendimento das mesmas. (classe InputStream e suas filhas)

8.5 - Para saber mais...

1-) Se eu não reescrever um método da minha classe mãe que é abstrato o código não irá compilar. Mas pode haver uma outra solução: posso declarar essa classe também abstrata!

2-) Uma classe que estende uma classe normal também pode ser abstrata! Ela não poderá ser instanciada, mas sua classe pai sim!

3-) Uma classe abstrata não precisa necessariamente ter um método abstrato.

8.6 - Exercícios

1-) Repare que a nossa classe `Conta` é uma excelente candidata para uma classe abstrata. Porque? Que métodos seriam interessantes candidatos a serem abstratos?

Transforme a classe `Conta` para abstrata, no main tente dar um `new` nela e compile o código.

```
abstract class Conta {
      // ...
```

```
}
```

2-) Se agora não podemos dar new em `Conta`, qual é a utilidade de ter um método que recebe uma referência a `Conta` como argumento? Aliás, posso ter isso?

3-) Remova o método `atualiza()` da `ContaPoupanca`, dessa forma ele herdará o método diretamente de `Conta`. Transforme o método `atualiza()` da classe `Conta` para abstrato. Compile o código. Qual é o problema com a classe `ContaPoupanca`?

```
abstract class Conta {

    // atributos e metodos que já existiam

    abstract void atualiza(double taxaSelic);
}
```

4-) Reescreva o método `atualiza()` na classe `Poupanca` para que a classe possa compilar normalmente.

5-) (opcional) Existe outra maneira da classe `ContaCorrente` compilar se você não reescrever o método abstrato?

6-) (opcional) Pra que ter o método atualiza na classe `Conta` se ele não faz nada? O que acontece se simplesmente apagamos esse método da classe `Conta`, e deixamos o método atualiza nas filhas?

7-) (opcional) Não podemos dar new em `Conta`, mas porque então podemos dar `new` em `Conta[10]`, por exemplo?

8-) (opcional) Você pode chamar o método atualiza de dentro da própria classe `Conta`? Porque?

capítulo 9

Orientação à Objetos – Interfaces

"O homem absurdo é aquele que nunca muda."
Georges Clemenceau -

Ao término desse capítulo, você será capaz de:
- dizer o que é uma interface e as diferenças entre herança e implementação;
- escrever uma interface em Java;
- utiliza-las como um poderoso recurso para diminuir acoplamento entre as classes.

9.1 - Aumentando nosso exemplo

Imagine que um Sistema de Controle do Banco pode ser acessado, além dos Gerentes, pelos Diretores do Banco. Então, teríamos uma classe `Diretor`:

```
class Diretor extends Funcionario {

	public boolean autentica(int senha) {
		// verifica aqui se a senha confere com a recebida como parametro
	}

}
```

E a classe `Gerente`:

```
class Gerente extends Funcionario {

	public boolean autentica(int senha) {
		// verifica aqui se a senha confere com a recebida como parametro
		// no caso do gerente verifica tambem se o departamente dele
		// tem acesso
	}

}
```

Repare que o método de autenticação de cada tipo de `Funcionario` pode variar muito. Mas vamos aos problemas. Considere o `SistemaInterno`, e seu controle, precisamos receber um `Diretor` ou `Gerente` como argumento, verificar se ele se autentica e coloca-lo dentro do sistema:

```
class SistemaInterno {

	void login(Funcionario funcionario) {
		// como chamo o método autentica? não posso! Nem todo Funcionario tem
	}
}
```

O SistemaInterno está aceitando qualquer tipo de `Funcionario`, ele tendo acesso ao sistema ou não, e nem todo `Funcionario` tem o método autentica, o que nos impede de chamar esse método com uma referência apenas a `Funcionario`. O que fazer então?

Uma possibilidade: criar dois métodos login no `SistemaInterno`: um para receber `Diretor` e outro para receber `Gerente`. Já vimos que essa não é uma boa escolha. Porque? Pois cada vez que criarmos uma nova classe de `Funcionario` que é *autenticável*, precisaríamos adicionar um novo método de login no `SistemaInterno`.

SOBRECARGA

Métodos com mesmo nome

Em Java, métodos podem ter o mesmo nome desde que não sejam ambíguos, isso é, que exista uma maneira de distinguir no momento da chamada.

Isso se chama **sobrecarga** de método. (**overloading**, não confundir com **overriding**, que é um conceito muito mais poderoso no caso).

Uma solução mais interessante seria criar uma classe no meio da árvores de herança, `FuncionarioAutenticavel`:

```
class FuncionarioAutenticavel extends Funcionario {

	public boolean autentica(int senha) {
		// faz autenticacao padrao
	}

	// outros atributos e metodos

}
```

As classes `Diretor` e `Gerente` passariam a estender de `FuncionarioAutenticavel`, e o `SistemaInterno` receberia referências desse tipo, como a seguir:

```
class SistemaInterno {

	void login(FuncionarioAutenticavel fa) {
		int senha = // captura senha de algum lugar, ou de um scanner de
polegar
		boolean ok = fa.autentica(senha);
		// aqui eu posso chamar o autentica!
		// Pois todo FuncionarioAutenticavel tem
	}
}
```

Repare que `FuncionarioAutenticavel` é uma séria candidata a classe abstrata. Mais ainda, o método `autentica` seria um método abstrato.

Esse caso herança resolve, mas vamos a uma outra situação:

Precisamos que todos os clientes também tenham acesso ao `SistemaInterno`. O que fazer? Uma opção é criar outro método `login` em `SistemaInterno`: descartamos essa. Uma outra, que é comum entre os novatos, é fazer uma herança sem sentido para resolver o problema, por exemplo fazer `Cliente extends` FuncionarioAutenticavel. Realmente resolve o problema, mas vai te trazer um monte de outros. `Cliente` definitivamente **não é** `FuncionarioAutenticavel`. Não faça herança quando a relação não é estritamente "é um".

Como resolver então?

9.2 - Interfaces

O que precisamos para resolver nosso problema? Arranjar uma forma de poder referenciar `Diretor`, `Gerente e Cliente` de uma mesma maneira. Isto é, achar um fator comum.

Se existisse uma forma que essas classes garantirem que possuem um determinado método, através de um contrato, resolveríamos o problema.

O contrato de um Autenticavel seria algo como: "A classe `Gerente` se compromete a ser tratada como `Autenticavel`, com isso, sendo obrigada a ter os métodos necessários".

Podemos criar esse contrato em Java!

```
interface Autenticavel {

      boolean autentica(int senha);

}
```

INTERFACE

Chama-se `interface` pois é a maneira a qual poderemos conversar com um `Autenticavel`. Interface é a maneira a qual conversamos com um objeto.

Uma interface pode definir uma série de métodos, mas nunca conter implementação deles. Ela só expõe **o que o objeto deve fazer**, e não **como ele faz**. Como ele faz vai ser definido em uma **implementação** dessa interface.

IMPLEMENTS

E o `Gerente` pode "assinar" o contrato, ou seja, **implementar** a interface. No momento que ele implementa essa interface, ele precisa escrever os métodos pedidos pela interface (muito próximo ao efeito de herdar métodos abstratos, aliás, métodos de uma interface são públicos e abstratos, sempre). Para implementar usamos a palavra chave `implements` na classe:

```
class Gerente extends Funcionario implements Autenticavel {

      private int senha;

      // outros atributos e métodos

      public boolean autentica(int senha) {
            if(this.senha != senha)
                  return false;

            // pode fazer outras possiveis verificacoes, como saber se esse
            // departamento do gerente tem acesso ao Sistema

            return true;
      }

}
```

A partir de agora, podemos tratar um `Gerente` como sendo um `Autenticavel`. Ganhamos polimorfismo! Temos mais uma forma de referenciar a um `Gerente`. Quando crio uma variável do tipo `Autenticavel`, estou criando uma referência para qualquer objeto de uma classe que implementa `Autenticavel`, direta ou indiretamente:

```
Autenticavel a = new Gerente();
// posso aqui chamar o metodo autentica!
```

Novamente, o proveito mais comum aqui seria receber como argumento. Voltamos ao nosso `SistemaInterno`:

```
class SistemaInterno {

      void login(Autenticavel a) {
            int senha = // captura senha de algum lugar, ou de um scanner de
polegar
            boolean ok = fa.autentica(senha);
            // aqui eu posso chamar o autentica!
            // não necessariamente é um Funcionario! Mais ainda, eu não sei
            // que objeto a referência "a" está apontando exatamente!
Flexibilidade.
      }
}
```

Pronto! E já podemos passar qualquer `Autenticavel` para o `SistemaInterno`. Então precisamos fazer com que o `Diretor` também implemente essa interface.

```
class Diretor extends Funcionario implements Autenticavel {

      // metodos e atributos, alem de obrigatoriamente ter o autentica

}
```

Você pode implementar mais de uma interface!

Diferentemente das classes, uma interface pode herdar de mais de uma interface. É como um contrato que depende de que outros contratos sejam fechados antes deste valer. Você não herda métodos e atributos, e sim responsabilidades.

Agora podemos passar um `Diretor`. No dia em que tivermos mais um funcionário com acesso ao sistema, basta que ele implemente essa interface, para se encaixar no sistema.

Ou se agora achamos que o `Fornecedor` precisa ter acesso: basta que ele implemente `Autenticavel`. Olhe só o tamanho do desacoplamento: quem escreveu o `SistemaInterno` só precisa saber que ele é `Autenticavel`, não faz diferença se é um `Diretor`, `Gerente`, `Cliente` ou qualquer classe que venha por aí. Basta seguir o contrato! Mais ainda, cada `Autenticavel` pode se autenticar de uma maneira completamente diferente de outro `Autenticavel`! A interface define que todos vão saber se autenticar (o que ele faz), a implementação define como exatamente vai ser feito (como ele faz).

A maneira como os objetos se comunicam num sistema orientado a objetos é muito mais importante do que como eles executam. **O que um objeto faz** é mais importante de **como ele faz**, seguindo essa regra seu sistema fica facil de dar manutenção, modificar, e muito mais!

9.3 - Dificuldade no aprendizado de interfaces

Interfaces representam uma barreira no aprendizado do Java: parace que estamos escrevendo um código que não serve pra nada, já que teremos essa linha (a assinatura do méotodo) escrita nas nossas classes implementadoras. Essa é uma maneira errada de se pensar. O objetivo do uso de uma interface é deixar seu código mais flexível, e possibilitar a mudança de implementação sem maiores traumas.

Os mais puristas dizem que toda classe deve ser "interfaceada", isto é, só devemos nos referir a objetos através de suas interfaces. Se determinada classe não

tem uma interface, ela deveria ter. Os autores acham tal medida radical demais, porém o uso de interfaces em vez de herança é amplamente aconselhado. (consultar os clássicos Design Patterns, Refactoring e Effective Java).

Veremos o uso de interfaces no capítulo de coleções, o que melhora o entendimento do assunto. O exemplo da interface Comparable também é muito esclarecedora.

9.4 - Um pouco mais...

1-) Posso substituir toda minha herança por interfaces? Qual é a vantagem e a desvantagem?

2-) Uma interface também pode declarar constantes (não atributos de objeto). Qual é a utilidade?

9.5 - Exercícios

1-) A sintaxe do uso de itnerfaces pode estranhas bastante a primeira vista. Vamos começar com um exercício para praticar a sintaxe:

```
interface AreaCalculavel{
	double calculaArea();
}
```

Queremos agora criar algumas classes que são `AreaCalculavel`:

```
class Quadrado implements AreaCalculavel {

	private int lado;

	Quadrado(int lado) {
		this.lado = lado;
	}

	public double calculaArea() {
		return this.lado;
	}
}

class Circulo implements AreaCalculavel {
	// ...
}

class Retangulo implements AreaCalculavel {
	// ...
}
```

Repare que aqui se você tivesse usado herança não ia ganhar muita coisa, já que cada implementação é totalmente diferente uma da outra: um Quadrado e um Circulo tem atributos e métodos **bem** diferentes, porém tem métodos em comum.

Mas mesmo que eles tivessem atributos em comum, utilizar interfaces é uma maneira muito mais elegante de modelar suas classes. Elas também trazem vantagems em não acoplar as classes (herança traz muito acoplamento, muitos autores clássicos dizem que em muitos casos **herança quebra o encapsulamento**, pensamento o qual os autores dessa apostila concordam plenamente).

2-) Nossos banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso vamos criar uma interface:

```
interface Tributavel {
```

```
        double calculaTributos(double taxa);
}
```

Alguns bens são tributáveis e outros não, por exemplo, repare que `ContaPoupanca` não é tributável:

```
class ContaCorrente extends Conta implements Tributavel {

        // para compilar o metodo calculaTributos precisa estar aqui
        // retorna um décimo da taxa vezes o saldo

}


class SeguroDeVida implements Tributavel {

        // para compilar o metodo calculaTributos precisa estar aqui
        // retorna a taxa vezes o saldo mais 10 reais.

}
```

Crie um `GerenciadorDeImpostoDeRenda` que recebe todos os tributáveis de uma pessoa e soma seus valores:

```
class GerenciadorDeImpostoDeRenda {
        private double taxa;
        private double total;

        GerenciadoDeImpostoDeRenda(double taxa) {
                this.taxa = taxa;
        }

        void adiciona(Tributavel t) {
                System.out.printl("Adicionando tributavel: " + t);

                // somar aqui o valor dos tributos ao total e imprimir o total atual
        }
}
```

Crie um `main` para instaciar diversas classes que implementam `Tributavel` e passar como argumento para um `GerenciadorDeImpostoDeRenda`. Repare que você não pode passar qualquer tipo de conta para o método adiciona, apenas a que implementa `Tributavel`. Além disso pode passar o `SeguroDeVida`.

3. (Opcional, Avançado) Transforme a classe `Conta` em uma interface. Atenção: faça isso num projeto a parte pois usaremos a `Conta` como classes nos exercícios futuros.

```
interface Conta {
        double getSaldo();
        void deposita(double valor);
        void retira(double valor);
        void atualiza(double taxaSelic);
}
```

Adapte `ContaCorrente` e `ContaPoupanca` para essa modificacao:

```
class ContaCorrente implements Conta {
        // ...
}

class ContaPoupanca implements Conta {
        // ...
}
```

Algum código vai ter de ser copiado e colado? Isso é tão ruim? Como você

poderia diminuir esse copia e cola e centralizar esses códigos repetidos em um lugar só? Pesquisar sobre henraça versus composição.

4. (Opcional) Subinterfaces:

As vezes é interessante criarmos uma interface que herda de outras interfaces. Dessa maneira quem for implementar essa nova interface precisa implementar todos os métodos herdades das suas superinterfaces (e talvez ainda novos métodos declarados dentro dela):

```
interface ContaTributavel extends Conta, Tributavel { }

class ContaCorrente implements ContaTributavel {
  // metodos
}

Conta c = new ContaCorrente();
Tributavel t = new ContaCorrente();
```

Repare que o código pode parecer estranho pois a interface não declara método algum, só herda os métodos abstratos declarados nas outras interfaces. Repare também que uma interface pode estender de mais de uma interface, sendo que classe só pode estender de uma (herança simples).

capítulo 10

Exceções – Controlando os erros

"Quem pensa pouco, erra muito"
Leonardo da Vinci -

Ao término desse capítulo, você será capaz de:
- controlar erros e tomar decisões baseadas nos mesmos;
- criar novos tipos de erros para sua melhorar o tratamento dos mesmos em sua aplicação ou biblioteca;
- assegurar que um método funcionou como diz em seu "contrato".

10.1 - Exceção

Voltando às `Conta`s que criamos no capítulo 6, o que iria acontecer ao tentar chamar o método `saca` com um valor fora do limite? O sistema iria mostrar uma mensagem de erro, mas quem chamou o método `saca` não irá saber que isso aconteceu.

Como avisar aquele que chamou o método que ele não conseguiu fazer aquilo que deveria?

Em Java, os métodos dizem qual o **contrato** que eles devem seguir, se ao tentar sacar ele não consegue fazer aquilo que deveria, ele precisa ao menos avisar o usuário que tentou sacar que isso não foi feito.

Veja no exemplo abaixo, estamos forçando uma `Conta` a ter um valor negativo, isto é, estar num estado inconsistente de acordo com a nossa modelagem.

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
minhaConta.saca(1000);
// o saldo é -900? É 100? É 0? A chamada ao método saca funcionou?
```

Em sistemas de verdade, é muito comum que quem saiba tratar o erro é aquele que chamou o método e não a própria classe! Portanto, nada mais natural que a classe sinalizar que um erro ocorreu.

A solução mais simples utilizada antigamente é a de marcar o retorno de um método como `boolean` e retornar `true` se tudo ocorreu da maneira planejada ou `false` caso contrário:

```
boolean saca(double quantidade) {
      if (quantidade > this.saldo + this.limite) { //posso sacar até saldo+limite
            System.out.println("Não posso sacar fora do limite!");
            return false;
      } else {
            this.saldo = this.saldo - quantidade;
            return true;
      }
}
```

Um novo exemplo de chamada ao método acima:

```
Conta minhaConta = new Conta();
minhaConta.setSaldo(100);
minhaConta.setLimite(100);
if (!minhaConta.saca(1000)) {
       System.out.println("Não saquei");
}
```

Mas e se fosse necessário sinalizar quando o usuário passou um valor negativo como **quantidade**? Uma solução é alterar o retorno de `boolean` para `int`, e retornar o código do erro que ocorreu. Isto é considerado uma má prática (conhecida também como uso de "magic numbers"). Além de você perder o retorno do método, o valor retornado é "mágico", e só legível perante extensa documentação, além de que não obriga o programador a tratar esse retorno, e no caso de esquecer isso seu programa continuará rodando.

Repare o que iria acontecer se fosse necessário retornar um outro valor. O exemplo abaixo mostra um caso onde através do retorno não será possível descobrir se ocorreu um erro ou não pois o método retorna um cliente.

```
public Cliente procuraCliente(int id) {

       if(idInvalido) {
              // avisa o método que chamou este que ocorreu um erro
       } else {
              Cliente cliente = new Client();
              cliente.setId(id);
              // cliente.setNome(……);
              return cliente;
       }

}
```

Por esse e outros motivos utilizamos um código diferente em Java para tratar aquilo que chamamos de exceções: os casos onde acontece algo que normalmente não iria acontecer. O exemplo do argumento do saque inválido ou do `id` inválido de um cliente é uma **exceção** a regra.

Exceção

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou errado no sistema.

10.2 - Matemático profissional?

Que tal tentar dividir um número por zero? Será que o computador consegue fazer aquilo que nós definimos que não existe?

```
public class TestandoADivisao {

       public static void main(String args[]) {
              int i = 5571;
              i = i / 0;
              System.out.println("O resultado é " + i);
       }
}
```

Tente executar o programa acima. O que acontece?

10.3 - Abusando de uma array

EXCEPTION

A maioria dos alunos já viu uma `Exception` de erro de acesso a uma array, mas vamos testar agora:

```
public void testaArray() {

	int nossaArray[] = new int[4];
	nossaArray[0] = 10;
	nossaArray[1] = 20;
	nossaArray[2] = 30;
	nossaArray[3] = 40;

	for(int i=0; i!=5; i++) {
		System.out.println(nossaArray[i]);
	}
}
```

```
C:\jce>java -cp . Teste
10
20
30
40
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at Teste.testaArray(Teste.java:17)
        at Teste.main(Teste.java:5)

C:\jce>
```

O laço `for` tenta acessar os itens 0, 1, 2, 3 e 4 da `nossaArray`, resultando em uma exceção do tipo `ArrayIndexOutOfBoundsException`.

TRY
CATCH

É possível tratar esse tipo de exceção com as palavras chave `try` e `catch`, que significam tente e pegue respectivamente.

```
public void testaArray() {

	int nossaArray[] = new int[4];
	nossaArray[0] = 10;
	nossaArray[1] = 20;
	nossaArray[2] = 30;
	nossaArray[3] = 40;

	try {
		for(int i=0; i!=5; i++) {
			System.out.println(nossaArray[i]);
		}
	} catch (ArrayOutOfBoundsException ex) {
		System.out.println("O erro " + ex.getMessage() + " ocorreu.");
	}
}
```

Se tentarmos entender o código acima:

1. o método `testaArray` cria uma array com quatro itens.
2. ele **tenta** acessar os itens da array, inclusive um item que não existe
3. ele **pega** uma exceção do tipo `ArrayOutOfBoundsException`, que é o objeto que demos o nome de `ex`, e mostra uma mensagem de erro na tela

Dica: Altere o laço `for` para que acesse somente os itens válidos, veja como o erro não ocorre e o programa flui normalmente!

Repare que um `ArrayIndexOutOfBoundsException` poderia ser facilmente evitado com o for corretamente escrito, ou com ifs que checariam os limites da array. Outros casos que também ocorrem exceções são quando você acessa uma referência nula (`NullPointerException`) ou quando faz um cast errado (veremos mais pra frente). Em todos os casos elas provavelmente poderiam ser evitadas pelo

programador. É por esse motivo que o java não te obriga a dar o try/catch nesses casos, e chamamos essas exceções de unchecked.

É interessante já passar para o exercício 1 deste capítulo, pois você vai exercitar a sintaxe, e verificar qual é a relação entre a tal pilha de execução e o controle de fluxo através do uso de Exceptions.

Erros

Os erros em Java são um tipo de exceção que também podem ser tratados. Eles representam problemas na máquina virtual e não devem ser tratados em 99% dos casos.

10.4 - Outro tipo de exceção: Checked Exceptions

Fica claro com os exemplos de código acima que não é necessário declarar que você está tentando fazer algo onde um erro possa ocorrer. Os dois exemplos, com ou sem o `try`/`catch`, compilaram e rodaram. Em um, o erro terminou o programa e no outro foi possível tratá-lo.

Mas não é só esse tipo de exceção que existe em Java, um outro tipo obriga os usuários que chamam o método ou construtor a tratar o erro. Um exemplo que podemos mostrar agora é o de abrir um arquivo para leitura, quando pode ocorrer o erro do arquivo não existir (veremos como trabalhar com arquivos em outro capítulo, não se preocupe com isto agora):

```
public static void metodo() {

       new java.io.FileReader("arquivo.txt");

}
```

O código acima não compila e o compilador avisa que é necessário tratar o `FileNotFoundException` que pode ocorrer:

```
C:\jce>javac Teste.java
Teste.java:5: unreported exception java.io.FileNotFoundException; must be caught
 or declared to be thrown
                java.io.FileReader leitor = new java.io.FileReader("arquivo.txt"
);
                                            ^
1 error
```

Para compilar e fazer o programa funcionar, precisamos tratar o erro de um de dois jeitos. O primeiro é tratá-lo com o `try` e `catch` do mesmo jeito que usamos no exemplo anterior com uma array:

```
public static void metodo() {

       try {
              new java.io.FileReader("arquivo.txt");
       } catch (java.io.FileNotFoundException fnfex) {
              System.out.println("Nao foi possivel abrir o arquivo para leitura");
       }

}
```

THROWS

A segunda forma de tratar esse erro é a de delegar ele para quem chamou o nosso método, isto é, passar para a frente.

```
public static void metodo() throws java.io.FileNotFoundException {
       new java.io.FileReader("arquivo.txt");
}
```

No início existe uma grande tentação de sempre passar o erro pra frente para outros tratarem dele. Pode ser que faça sentido dependendo do caso mas não até o main, por exemplo. Acontece que quem tenta abrir um arquivo sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa pode não saber ou, pior ainda, tentar abrir cinco arquivos diferentes e não saber qual deles teve um problema!

Não há uma regra para decidir em que momento do seu programa você vai tratar determinar exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação a aquele erro. Enquanto não for o momento, você provavelmente vai preferir delegar a responsabilidade para o método que te invocou.

10.5 - Mais de um erro

É possível tratar mais de um erro quase que ao mesmo tempo:

1. Com o `try` e `catch`:

```
try {
       objeto.metodoQuePodeLancarIOeSQLException();
} catch (IOException e) {
       // ..
} catch (SQLException e) {
       // ..
}
```

2. Com o `throws`:

```
public void abre(String arquivo) throws IOException, SQLException {
       // ..
}
```

Você pode também escolher tratar algumas exceções e declarar as outras no throws:

```
try {
       // ...
} catch (IOException e) {
       // ..
} catch (SQLException e) {
       // ..
}
```

3. Com o `throws`:

```
public void abre(String arquivo) throws IOException {
       try {
              objeto.metodoQuePodeLancarIOeSQLException();
       } catch (SQLException e) {
              // ..
       }
}
```

É desnecessário declarar no throws as exceptions que são unchecked, porém é permitido e as vezes facilita a leitura e a documentação do seu código.

10.6 - E finalmente...

FINALLY

Os blocos `try` e `catch` podem conter uma terceira cláusula chamada `finally`

que indica o que deve ser feito após o término do bloco `try` ou de um `catch` qualquer.

No exemplo a seguir, o bloco `finally` será executado não importa se tudo ocorrer ok ou com algum problema:

```
try {
      // bloco try
} catch (IOException ex) {
      // bloco catch 1
} catch (SQLException sqlex) {
      // bloco catch2
} finally {
      // bloco finally
}
```

10.7 - Criando novas exceções

Podemos também lançar uma exception, o que é extramamente útil. Dessa maneira resolvemos o problema de alguém poder esquecer de fazer um if no retorno de um método.

Considere o exemplo de um carro que não pode ultrapassar uma determinada velocidade:

```
public class Carro {

      private double velocidade;
      private double velocidadeMaxima = 120;
      // ...

      void acelera(double valor) {
            if (valor + this.velocidade > this.velocidadeMaxima) {
                  throw new RuntimeException();
            } else {
                  this.velocidade += valor;
            }
      }
}
```

A palavra chave **throw** lança uma Exception (diferente de throws, que apenas avisa da possibilidade daquele método lança-la). No nosso caso lança uma do tipo unchecked. RuntimeException é a exception mãe de todas as exceptions unchecked.

A desvantagem aqui é que ela é muito genérica, quem receber esse erro não sabe dizer exatamente qual foi o problema. Podemos então usar uma Exception mais específica:

```
      void acelera(double valor) {
            if (valor + this.velocidade > this.velocidadeMaxima) {
                  throw new IllegalArgumentException();
            } else {
                  this.velocidade += valor;
            }
      }
```

IllegalArgumentException diz um pouco mais: algo foi passado como argumento e seu método não gostou. Ela é uma Exception unchecked pois estende de RuntimeException e já faz parte da biblioteca do java. (IllegalArgumentException é melhor de ser usado quando um argumento sempre é inválido, como por exemplo números negativos, referências nulas, etc).

Podíamos melhorar ainda mais e passar para o construtor da

IllegalArgumentException o motivo da excecao:

```
	void acelera(double valor) {
		if (valor + this.velocidade > this.velocidadeMaxima) {
			throw new IllegalArgumentException("voce acelerou mais do que
devia!");
		} else {
			this.velocidade += valor;
		}
	}
```

O método getMessage() definido na classe Throwable (mae de todos os tipos de erros e exceptions) vai retonar a mensagem que passamos ao construtor da IllegalArgumentException.

É bem comum criar uma própria classe de exceção para controlar melhor o uso de suas exceções, dessa maneira podemos passar valores específicos para ela carregar, e que sejam úteis de alguma forma. Vamos criar a nossa:

```
public class VelocidadeUltrapassadaException extends RuntimeException {

	public VelocidadeUltrapassadaException(String message) {
		super(message);
	}

}
```

THROW

E, para usá-la na nossa classe Carro, novamente usamos a palavra chave `throw`, que irá "jogar" a nova exceção.

```
public class Carro {

	private double velocidade;
	private double velocidadeMaxima = 120;
	// ...

	void acelera(double valor) {
		if (valor + this.velocidade > this.velocidadeMaxima) {
			throw new VelocidadeUltrapassadaException("ultrapassou
limite!");
		} else {
			this.velocidade += valor;
		}
	}
}
```

Agora, falta usar o código para verificar o erro usando exceções:

```
Carro c = new Carro();
c.acelera(100);
c.acelera(100);
```

Podemos transformar essa Exception de unchecked para checked, obrigando assim quem chama esse método a dar try-catch, ou throws:

O código que esta chamando acelera agora não compila! Precisamos dar throws ou try catch:

```
Carro c = new Carro();
try {
	c.acelera(100);
	c.acelera(100);
```

```
} catch (VelocidadeMaximaException ex) {
      System.out.println("problemas: " + ex.getMessage());
}
```

Poderíamos ainda mexer na nossa VelocidadeUltrapassadaException para que ela guardasse algo mais útil que uma simples mensagem: o valor da velocidade que você estava querendo atingir, sendo que você tinha um máximo:

```
public class VelocidadeUltrapassadaException extends RuntimeException {

      private double valor;
      private double maximo;

      public VelocidadeUltrapassadaException(double valor, double maximo) {
            this.valor = valor;
            this.maximo = maximo;
      }

      // getters para valor e maximo!

}
```

Agora no momentod e criar a VelocidadeUltrapassadaException devemos passar os argumentos necessários:

```
      void acelera(double valor) {
            if (valor + this.velocidade > this.velocidadeMaxima) {
                  throw new VelocidadeUltrapassadaException(valor +
this.velocidade, this.velocidadeMaxima);
            } else {
                  this.velocidade += valor;
            }
      }
```

Dentro do nosso bloco de catch, podemos chamar getValor e getMaximo para explicar melhor o ocorrido. Ou então podemos reescrever o método getMessage da VelocidadeUltrapassadaException:

```
public class VelocidadeUltrapassadaException extends RuntimeException {

      private double valor;
      private double maximo;

      public VelocidadeUltrapassadaException(double valor, double maximo) {
            this.valor = valor;
            this.maximo = maximo;
      }

      // getters para valor e maximo!

      public String getMessage() {
            return "Voce tentou ir para a velocidade " + this.valor + " sendo que
sua velocidade maxima era " + this.maximo;
      }
}
```

catch e throws

Existe uma péssima prática de programação em java que é a de escrever o `catch` e o `throws` com `Exception`.

Existem códigos que sempre usam `Exception` pois isso cuida de todos os possíveis erros. O maior problema disso é generalizar o erro. Se alguém joga algo do tipo

`Exception` para quem o chamou, quem recebe não sabe qual o tipo específico de erro ocorreu e não vai saber como tratar o mesmo.

10.8 - Um pouco mais...

1-) É possível criar sua própria exceção de tempo de execução (não checked), basta utilizar RuntimeException em vez de `Exception`.

2-) É possível criar um bloco `try` e `finally`, sem catch. Isso significa se não ocorrer erro algum, o bloco do `finally` irá ser executado. Se ocorrer algum erro, ele também será executado, e o erro irá ser jogado para quem chamou o método.

3-) Procure informações sobre a palavra-chave `assert` e tente utilizar o código abaixo:

```
int i = 1;
// tenho certeza que i vale 1
assert i == 1;
i = 2;
// tenho certeza que i vale 2
assert i == 2;
i++;
// vai gerar um erro pois i não vale 4
assert i == 4;
```

Ele só irá funcionar se compilado com a opção **–source 1.4** e rodado com **–ea** (enable assertions).

10.9 - Exercícios

1-) Teste o seguinte código você mesmo:

```
class Teste {
	public static void main(String[] args) {
		System.out.println("inicio do main");
		metodo1();
		System.out.println("fim do main");
	}

	public static void metodo1() {
		System.out.println("inicio do metodo1");
		metodo2();
		System.out.println("fim do metodo1");
	}

	public static void metodo2() {
		System.out.println("inicio do metodo2");
		int[] array = new int[10];
		for(int i = 0; i <= 15; i++) {
			array[i] = i;
			System.out.println(i);
		}
		System.out.println("fim do metodo2");
	}
}
```

Rode o código. Leia a stacktrace. O que ela indica?

2-) Adicione um `try/catch` em volta do `for`, pegando `ArrayIndexOutOfBoundsException`. O que o código imprime agora?

```
try {
      for(int i = 0; i <= 15; i++) {
            array[i] = i;
            System.out.println(i);
      }
} catch (ArrayIndexOutOfBoundsException e) {
      System.out.printl("erro: " + e);
}
```

Em vez de fazer o `try` em torno do for inteiro, tente apenas com o bloco de dentro do for:

```
for(int i = 0; i <= 15; i++) {
      try {
            array[i] = i;
            System.out.println(i);
      } catch (ArrayIndexOutOfBoundsException e) {
            System.out.printl("erro: " + e);
      }
}
```

Qual é a diferença?

Agora retire o `try/catch` e coloque ele em volta da chamada do `metodo2`. Faca a mesma coisa, retirando o `try/catch` novamente e colocando em volta da chamada do `metodo1`. Rode os códigos, o que acontece?

Repare que a partir do momento que uma exception foi "catched" (tratada, handled), a execução volta ao normal a partir daquele ponto.

3-) Na classe `Conta`, modifique o método `saca(double x)` para que ele agora retorne `void`, e quando não houver saldo suficiente, ele deve lançar uma exception chamada `IllegalArgumentException`, que já faz parte da biblioteca do java e é muito utilizada em casos como esse.

4-) Modifique a sua main para que ela faça um loop em uma conta e va sacando dinheiro até não haver mais. Algo como:

```
ContaCorrente c = new ContaCorrente();
c.deposita(1000);
for(int i = 0; i < 20; i++) {
      c.saca(100);
}
```

Repare na stacktrace. Perceba as informações específicas que ela te traz: é um snapshot da pilha no momento em que a exception foi criada.

Brinque com o `try catch` para conter a exceção.

Existe agora alguma forma do programador esquecer de verificar esse problema? Qual a vantagem dessa abordagem em relação ao do uso do `boolean` como retorno?

5-) Crie sua própria `Exception`, `SaldoInsuficienteException`. Para isso você precisa criar uma classe com esse nome que extenda de `RuntimeException`. Lance-a em vez de `IllegalArgumentException`.

```
class SaldoInsuficienteException extends RuntimeException {

}
```

6-) (opcional) Coloque um construtor na classe `SaldoInsuficienteException` que receba o quanto ficou faltando de dinheiro para a operação se completar. Dessa maneira, na hora de dar o throw new `SaldoInsuficienteException` você vai precisar passar a diferença de valores como argumento para o construtor.

7-) (opcional) A classe `Exception` (na verdade `Throwable`) tem definida nela um método `getMessage` que pode ser reescrito nas suas subclasses para falar um pouco mais sobre o erro. Reescreva-o na sua classe `SaldoInsuficienteException` para algo como:

```
public String getMessage() {
      return "Ficou faltando " + valor + " reais para a operacao ser bem sucedida".
}
```

Agora ao imprimir a exception, repare que essa mensagem é que vai aparecer.

8-) (opcional) Declare a classe `SaldoInsuficienteException` como filha de `Exception` em vez de `RuntimeException`. O que acontece?

Você vai precisar avisar que o seu método `retira() throws SaldoInsuficienteException`, pois ela é uma checked exception. Além disso, quem chama esse método vai precisar tomar uma decisão entre `try-catch` ou `throws`.

10.10 - Desafios

1-) O que acontece se acabar a memória do java? Como forçar isso?

capítulo 11

Pacotes – Organizando suas classes e bibliotecas

"Uma discussão prolongada significa que ambas as partes estão erradas"
Voltaire -

Ao término desse capítulo, você será capaz de:
- separar suas classes em pacotes e
- preparar arquivos simples para distribuição.

11.1 - Organização

Quando um programador utiliza as classes feitas por outro surge um problema clássico: como escrever duas classes com o mesmo nome?

Por exemplo, pode ser que a minha classe de `Data` funcione de um certo jeito e a classe de `Data` de um colega de outro jeito. Pode ser que a classe de `Data` de uma **biblioteca** funcione ainda de terceira maneira.

Como permitir que tudo isso realmente funcione? Como controlar quem quer usar qual classe de `Data`? A solução é simples: criar diretórios para organizar as classes.

PACOTE

Os diretórios estão diretamente relacionados aos chamados **pacotes** e costumam agrupar classes de funcionalidade parecida.

No pacote `java.util`, por exemplo, temos as classes `Date`, `SimpleDateFormat` e `GregorianCalendar`; todas elas trabalham com datas de formas diferentes.

Se a classe `Cliente` está no pacote `banco`, ela deverá estar no diretório `banco`. Se ela se localiza no pacote `br.com.caelum.banco`, significa que está no diretório **br/com/caelum/banco**.

A classe `Cliente` que se localiza nesse último diretório mencionado deve ser escrita da seguinte forma:

```
package br.com.caelum.banco;

class Cliente {
      // ...
}
```

PACKAGE

Fica fácil notar que a palavra chave `package` indica qual o pacote que contém esta classe.

Um pacote pode conter nenhum, um ou mais subpacotes e/ou nenhuma, uma ou mais classes dentro dele.

Padrão da nomenclatura dos pacotes

O padrão da sun para dar nome aos pacotes é relativo ao nome da empresa que desenvolveu a classe:

```
br.com.nomedaempresa.nomedoprojeto.subpacote
br.com.nomedaempresa.nomedoprojeto.subpacote2
br.com.nomedaempresa.nomedoprojeto.subpacote2.subpacote3
```

Os pacotes só possuem letras minúsculas, não importa quantas palavras estejam contidas nele. Esse padrão existe para evitar ao máximo o conflito de pacotes de empresas diferentes.

As classes do pacote padrão de bibliotecas não seguem essa nomenclatura, que foi dada para bibliotecas de terceiros.

11.2 - Import

Para usar uma classe do mesmo pacote, basta fazer referência a ela como foi feito até agora, simplesmente escrevendo o próprio nome da classe. Se existe uma classe `Banco` dentro do pacote `br.com.caelum.banco`, ela deve ser escrita assim:

```
package br.com.caelum.banco;

class Banco {
	String nome;
	Cliente clientes[];
}
```

A classe `Cliente` deve ficar no mesmo pacote da seguinte maneira:

```
package br.com.caelum.banco;

class Cliente {
	String nome;
	String endereco;
}
```

A novidade chega ao tentar utilizar a classe `Banco` (ou `Cliente`) em uma outra classe que esteja em outro pacote, por exemplo no pacote `br.com.caelum.util`:

```
package br.com.caelum.banco.util;

class TesteDoBanco {

	public static void main(String args[]) {
		br.com.caelum.banco.Banco meuBanco = new br.com.caelum.banco.Banco();
		meuBanco.nome = "Banco do Brasil";
		System.out.println(meuBanco.nome);
	}

}
```

Repare que precisamos referenciar a classe `Banco` com todo o nome do pacote na sua frente.

Mesmo assim, ao tentar compilar a classe anterior o compilador irá reclamar que a classe `Banco` não está visível.

Acontece que as classes só são visíveis para outras no **mesmo pacote** e para permitir que a classe `TesteDoBanco` veja e acesse a classe `Banco` em outro pacote precisamos alterar essa última e transformá-la em pública:

```
package br.com.caelum.banco;
```

```
public class Banco {
       String nome;
       Cliente clientes[] = new Cliente[2];
}
```

A palavra chave `public` libera o acesso para classes de outros pacotes. Do mesmo jeito que o compilador reclamou que a classe não estava visível, agora ele reclama que o nome também não está. É fácil deduzir como resolver o problema, algo que já foi visto anteriormente:

```
package br.com.caelum.banco;

public class Banco {
       public String nome;
       public Cliente clientes[] = new Cliente[2];
}
```

Agora já podemos testar nosso exemplo anterior.

Voltando ao código do `TesteDoBanco`, é necessário escrever todo o pacote para identificar qual classe queremos usar? O exemplo que usamos ficou bem complicado de ler:

```
br.com.caelum.banco.Banco meuBanco = new br.com.caelum.banco.Banco();
```

IMPORT

Existe uma maneira mais simples de se referenciar a classe `Banco`: basta **importá-la** do pacote `br.com.caelum.banco`:

```
package br.com.caelum.banco.util;

import br.com.caelum.banco.Banco; // agora podemos nos referenciar
                                  // a Banco diretamente

class TesteDoBanco {

       public static void main(String args[]) {
              Banco meuBanco = new Banco();
              meuBanco.nome = "Banco do Brasil";
       }

}
```

package, import, class

É muito importante manter a ordem! Primeiro aparece uma (ou nenhuma) vez o package, depois pode aparecer um ou mais import e por último as declarações de classes.

import x.y.z.*;

É possível importar um pacote inteiro (todas as classes do pacote, **exceto os subpacotes**) através do *:

```
import java.util.*;
```

Importar todas as classes de um pacote não implica em perda de performance em tempo de execução mas pode trazer problemas com classes de mesmo nome! Além disso, importar de um em um é considerado boa prática pois facilita a leitura para outros programadores

11.3 - Import Estático

Algumas vezes, escrevemos classes que contém muitos métodos e atributos estáticos (finais, como constantes). Essas classes são classes utilitárias, e precisamos sempre nos referir a elas antes de chamar um método ou utilizar um atributo:

```
import pacote.ClasseComMetodosEstaticos;
class UsandoMetodosEstaticos {
       void metodo() {
              ClasseComMetodosEstaticos.metodo1();
              ClasseComMetodosEstaticos.metodo2();
       }
}
```

STATIC IMPORT

Começa a ficar muito chato de escrever toda hora o nome da classe. Para resolver esse problema, no Java 5.0 foi introduzido o `static import`, que importa métodos e atributos estáticos de qualquer classe. Usando essa nova técnica, você pode importar os métodos do exemplo anterior e usá-los diretamente:

```
import static pacote.ClasseComMetodosEstaticos.*;
class UsandoMetodosEstaticos {
       void metodo() {
              metodo1();
              metodo2();
       }
}
```

Apesar de você ter importado todos os métodos e atributos estáticos da classe `ClasseComMetodosEstaticos`, a classe em si não foi importada, e se você tentasse der `new`, por exemplo, ele não ia conseguir encontrá-la, precisando de um `import` normal a parte.

Um bom exemplo de uso são os métodos e atributos estáticos da classe de matemática do Java.

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

class TesteMatematico {
       double areaDaCircunferencia (double raio) {
              return PI * raio * raio; // usamos PI ao invés de Math.PI !!
       }
}
```

11.4 - Acesso aos atributos, construtores e métodos

Os modificadores de acesso existentes em Java são quatro sendo que até o momento já vimos três, mas só explicamos dois.

A diferença entre eles é descrita a seguir:

`public` – Todas as classes podem acessar aquilo que for definido como `public`. Classes, atributos, construtores e métodos podem ser `public`.

`protected` – Aquilo que é `protected` pode ser acessado por todas as classes do mesmo pacote e por todas as classes que a estendam. Somente atributos, construtores e métodos podem ser `protected`.

padrão (sem nenhum modificador) – Se nenhum modificador for utilizado, todas as classes do mesmo pacote têm acesso ao atributo, construtor, método ou classe.

`private` – A única classe capaz de acessar os atributos, construtores e métodos privados é a própria classe. Classes não podem ser `private`, mas atributos,

construtores e métodos sim.

Classes públicas

Para melhor organizar seu código, o Java não permite mais de uma classe pública por arquivo e o arquivo deve ser **NomeDaClasse.java**.

Uma vez que outros programadores irão utilizar essa classe, quando precisarem olhar o código da mesma, fica mais fácil encontrá-la sabendo que ela está no arquivo de mesmo nome.

11.5 - Arquivos, bibliotecas e versões

Assim que um programa fica pronto, é meio complicado enviar dezenas ou centenas de classes para cada cliente que quer utilizá-lo.

JAR

O jeito mais simples de trabalhar com um conjunto de classes é compactá-los em um arquivo só. O formato de compactação padrão é o **ZIP**, porém a extensão do arquivo compactado será **JAR**.

O arquivo .jar

O arquivo **jar** possui uma série de classes (e arquivos de configurações) compactados, no estilo de um arquivo **zip**. O arquivo **jar** pode ser criado com qualquer compactador **zip** disponível no mercado, inclusive o programa **jar** que vem junto com o **sdk**.

Para criar um arquivo jar do nosso programa de banco, basta ir ao diretório onde estão contidas as classes e usar comando a seguir para criar o arquivo **banco.jar** com todas as classes dos pacotes `br.com.caelum.util` e `br.com.caelum.banco`.

```
jar –cvf banco.jar br/com/caelum/util/*.class br/com/caelum/banco/*.class
```

Para usar esse arquivo **banco.jar** para rodar o `TesteDoBanco` basta rodar o java com o arquivo jar como argumento:

```
java –classpath banco.jar br.com.caelum.util.TesteDoBanco
```

Para adicionar mais arquivos .**jar**, que podem ser bibliotecas, ao programa basta rodar o java da seguinte maneira:

```
java –classpath biblioteca1.jar;biblioteca2.jar NomeDaClasse
```

Vale lembrar que o ponto e vírgula utilizado só é válido em ambiente windows e depende de cada sistema operacional (no linux é o dois pontos).

Bibliotecas

Diversas bibliotecas podem ser controladas de acordo com a versão por estarem sempre compactadas em um arquivo .jar. Basta verificar o nome da biblioteca (por exemplo **gujChat0.9.8.jar**) para descobrir a versão dela.

Então é possível rodar dois programas ao mesmo tempo, cada um utilizando uma versão da biblioteca através do parâmetro **–classpath** do java.

Criando um .jar automaticamente

Existem diversas ferramentas que servem para automatizar o processo de deploy, que

consiste em compilar, gerar documentação, bibliotecas etc. As duas mais famosas são o **ANT** e o **MAVEN**, ambos são projetos do grupo Apache.

11.6 - Um pouco mais...

1-) Classes internas também tem acesso aos atributos privados da classe que a engloba. Classes internas não foram ainda discutidas e são um ponto de difícil entendimento no Java.

11.7 - Exercícios

1-) Escolha um dos seus sitemas: Conta ou Funcionario, e passe ela para pacotes. Respeite a conveção de código da sun, por exemplo:

br.com.empresa.banco: colocar classe com o main aqui
br.com.empresa.banco.contas : colocar Conta e suas filhas aqui
br.com.empresa.banco.sistema : colocar AtualizadorDeContas aqui

2-) Coloque cada classe em seu respectivo arquivo .java. Independente se ela será publica (boa prática).

3-) O código não vai compilar prontamente, pois muitos métodos que declaramos são package-friendly, quando na verdade precisaríamos que eles fossem public. O mesmo vale para as classes, algumas delas precisarão ser publicas.

4-) (opcional) Gere um jar do seu sistema, com o arquivo de manifesto. Execute-o com java -jar.

5-) (opcional) Gere o javadoc do seu sistema.

11.8 - Desafios

1-) Crie um sistema de banco e cliente dividido por pacotes e com o devido acesso aos tipos de variáveis.

capítulo 12

O pacote padrão

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

Francis Piacaba -

Ao término desse capítulo você será capaz de:

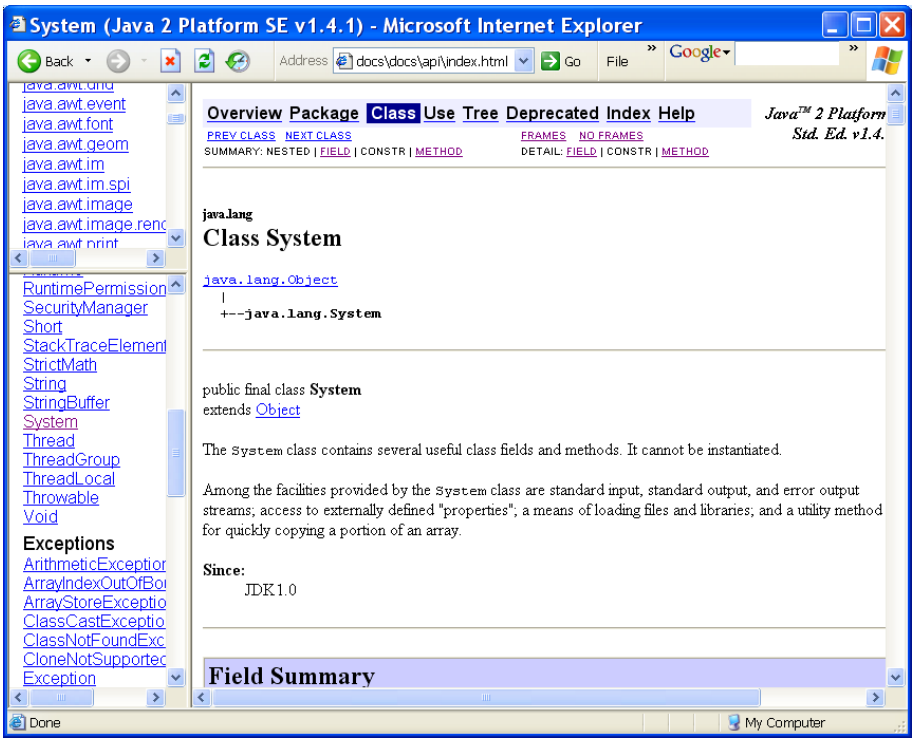
- utilizar as principais classes do pacote java.lang e ler a documentação padrão de projetos java: o javadoc;
- usar a classe System para obter informações do sistema;
- utilizar a classe String de uma maneira eficiente e conhecer seus detalhes;
- usar as classes wrappers (como Integer) e boxing e
-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

12.1 - Documentação do Java

Como vamos saber o que cada classe tem no Java? Quais são seus métodos, o que eles fazem?

JAVADOC

No site da Sun, você pode também estar baixando a documentação das bibliotecas do Java, freqüentemente chamada de "**javadoc**".



Nesta documentação, no quadro superior esquerdo você encontra os pacotes, e o inferior esquerdo está a listagem das classes e interfaces do respectivo pacote.

Clicando-se em uma classe ou interface, o quadro da direita passa a detalhar todos atributos e métodos não privados (e porque será que não são mostrados os privados?).

12.2 - Pacote padrão

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.

12.3 - Um pouco sobre a classe System e Runtime

A classe `System` possui uma série de atributos e métodos estáticos. Já usamos o atributo `System.out`, para imprimir. Ela também possui o atributo `in`, que lê da entrada padrão alguns 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`. Será útil ficar lendo de byte em byte?

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);
```

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");
```

É desnecessário dizer que isto deve ser evitado ao máximo, já que gera uma dependência da sua aplicação com o sistema operacional em questão, pedendo a portabilidade. Em muitos casos isso pode ser substituído por chamadas as bibliotecas do java, esse caso por exemplo você tem um método list na clase 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.

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

12.4 - 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 {

}
```

OBJECT

Quando o Java não encontra a palavra chave `extends`, ele então 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 a mesma coisa:

```
class MinhaClasse extends Object {

}
```

Todas as classes, sem exceção, herdam de `Object`, seja direta ou indiretamente.

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**.

12.5 - Casting de referências

A habilidade de poder se referenciar a qualquer objeto como `Object` nos traz muitas vantagens. Podemos criar um método que recebe um `Object` como argumento, isto é, qualquer coisa! Melhor, podemos armazenar qualquer objeto:

```
class GuardadorDeObjetos {
	private Object[] array = 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.array[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);

// ...

Object object = guardador.pegaObjeto(0); // pega a conta referenciado como objeto
object.getSaldo(); // classe Object nao tem método getSaldo! não compila!
```

Poderíamos então atribuir essa referência de `Object` para `Conta`? Tentemos:

```
Conta contaResgatada = object;
```

Nós temos certeza de que esse `Object` se referencia a uma `Conta`, já que fomos nós que o adicionamos na classe que guarda objetos. Mas o Java não tem garantias sobre isso! Essa linha acima não compila, pois nem todo `Object` é uma `Conta`.

CASTING DE REFERÊNCIAS

Para conseguir realizar essa atribuição, devemos "avisar" o Java que realmente queremos fazer isso, sabendo do risco que podemos estar correndo. Fazemos o **casting de referências**, parecido como quando fizemos com os tipos primitivos:

```
Conta contaResgatada = (Conta) object;
```

Agora o código passa a compilar, mas será que roda? Esse código roda sem

nenhum problema, pois em tempo de execução ele irá verificar se essa referência realmente está se referindo a um `Carro`, e está! Se não tivesse, 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 tenha o casting, 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;
```

Atenção! O problema aqui poderia ser resolvido através da parametrização da classe `GuardadorDeObjetos`. Veja o apêndice de **generics**.

12.6 - Integer

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?".

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`).

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.

WRAPPING

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

```
int i = 5;
Integer x = new Integer(i);
```

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();
```

12.7 - Autoboxing no Java 5.0

AUTOBOXING

Esse processo de wrapping e unwrapping é entediante. O Java 5.0 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 ele compila perfeitamente. É importante ressaltar que isso não quer dizer que tipos primitivos e referências agora são a mesma coisa, isso é simplesmente um "adocicamente sintático" para facilitar a codificação.

Mas isso não é um simples wrapping:

```
Integer x = 5;
Integer y = 5;
System.out.println(x == y);
```

Aqui o retorno é true, diferentemente se você tivesse feito o wrapping com `new Integer(5)`. Isso nos da uma certa segurança em usar auto boxing, já que não estaremos criando novos objetos na memória todo tempo. Mas repare que para número grandes isso já não vale:

```
Integer x = 5000;
Integer y = 5000;
System.out.println(x == y);
```

Então tome cuidado ao fazer autoboxings desnecessários, em um for grande a alocação de um grande número de wrappers pode estressar a máquina virtual.

Você pode fazer o autoboxing diretamente para `Object` também:

```
Object o = 5;
```

12.8 - Alguns métodos do java.lang.Object

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:

```
public 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` agora:

```
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 `String`s 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("descricao: " + c);
```

Gera o mesmo resultado! Você ainda pode concatenar `String`s em Java como o operador `+`. Se o Java encontra um objeto no meio da concatenação, ele também chama o `toString` dele.

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!");
}
```

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`.

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 o `==` 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 chamam ele através do polimorfismo, como veremos no capítulo do `java.util`.

O método `hashCode()` anda de mão dada 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 falamos dele no capítulo de `java.util`.

12.9 - java.lang.String

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

Aliás, podemos criar uma `String` utilizando-se do `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("mesmo objeto");
}
else {
	System.out.println("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` a `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 `String`s usando o `+`. Podemos concatenar `String`s com qualquer outra coisa, até mesmo objetos e números:

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

SPLIT

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

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

COMPARETO

Se quisermos comparar duas `String`s, 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, se ela não fosse imutável, mudando o valor de uma String afetaria nas Strings que outras classes estão se referindo e tem o mesmo valor.

Repare o código abaixo:

```
String palavra = "fj11";
palavra.toUpperCasa();
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. O código realmente útil ficaria assim:

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

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

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

Isso funciona para todos os métodos da classe String, seja replace, trim, toLoweCase e outros.

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 também no seu método intern().

charAt e length

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

Por exemplo o método `charAt(int)` retorna o caractere existente na posição `i` da string e o método `length` retorna o número de caracteres na mesma.

12.10 - java.lang.Math

Na classe `Math`, existe uma série de métodos estáticos que fazem operações com número, como por exemplo arredondar, tirar o valor absoluto, tirar a raiz, calcular o seno e outros.

```
double d = 4.6;
int 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 `static import` aqui:

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

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

```
double d = 4.6;
int i = round(d);

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

12.11 - Um pouco mais...

1-) Você pode gerar a documentação das suas classes, neste mesmo formato da Sun. Essa ferramenta vem junto com o JSDK e chama-se javadoc. Aprenda a utilizá-la. Onde você deve escrever o conteúdo que vai aparecer nessa documentação? Pesquise.

2-) A String possui o método `compareTo` pois está definido em uma das interfaces que ela implementa, no caso a `Comparable`. Essa interface tem muitos usos, entre os quais para ordenar um monte de objetos. Leia sobre ela.

12.12 - Exercícios

1-) Teste os exemplos desse capítulo, para ver que uma `String` é imutável.

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

3-) Reescrava o método toString da sua classe `Conta` e imprima uma referência para Conta. Remova esse método, se você tentar imprimir uma referência a `Conta` o que aparece?

4-) Reescreva o método `equals` da classe `Conta` para que duas contas com o mesmo número de conta sejam consideradas iguais. Compare duas instâncias de `Conta` com ==, depois com `equals`, sendo que as instâncias são diferentes mas possuem o mesmo número.

5-) Um double não está sendo suficiente para eu guardar a quantidade de casas necessárias em uma aplicação. Preciso guardar um número decimal muito grande! O que poderia usar? (consulte a **documentação**, tente adivinhar onde você pode encontrar algo desse tipo pelos nomes dos pacotes, veja como é intuitivo).

6-) (opcional) Agora faça com que o `equals` da sua classe `Conta` também leve em consideração a `String` do nome do cliente a qual ela pertence. Teste-a.

7-) (opcional) Escreva um método que usa os métodos `charAt` e `length` de uma `String` para imprimir a mesma caractere a caractere, sendo que cada caractere deve estar em uma linha diferente.

8-) (opcional) Reescreva o método do exercício quatro mas agora imprima a string de trás para a frente.

9-) (opcional) Crie a classe GuardadorDeObjetos desse capítulo. Teste-a adicionando uma String e depois retirnando com o casting para uma outra classe, como Conta. Repare a exception que é lançada. Teste também o autoboxing do java 5.0.

10-) (opcional) Pesquise a classe StringBuilder (ou StringBuffer no java 1.4). Ela é mutável. Porque usá-la em vez da String? Quando usá-la?

12.13 - Desafio

Converta uma String para um número, sem usar as bibliotecas do java que ja fazem isso. Isso é, uma String x = "767" deve gerar um int i = 767.

capítulo 13

Pacote java.io

"A beneficência é sobretudo um vício do orgulho e não uma virtude da alma."
Doantien Alphonse François (Marquês de Sade) -

Ao término desse capítulo, você será capaz de:
- ler e escrever bytes, caracteres e Strings de/para a entrada e saída padrão;
- ler e escrever bytes, caracteres e Strings de/para arquivos e
- utilizar buffers para agilizar a leitura e escrita através de fluxos.

13.1 - Orientação a objeto

JAVA.IO

Assim como todo o resto das bibliotecas em Java, a parte de controle de entrada e saída de dados (conhecido como **io**) é orientada a objetos e usa os principais conceitos mostrados até agora: interface, classes abstratas e polimorfismo.

ARQUIVOS
SOCKETS
ENTRADA E SAÍDA

A idéia atrás do polimorfismo no pacote java.io é de utilizar fluxos de entrada (`InputStream`) e de saída (`OutputStream`) para toda e qualquer operação, seja ela relativa a um **arquivo**, a uma conexão remota via **sockets** ou até mesmo a **entrada** e **saída padrão** de um programa (normalmente o teclado e o console).

As classes abstratas `InputStream` e `OutputStream` definem respectivamente o comportamento padrão dos fluxos em Java: em um fluxo de entrada é possível ler bytes e no fluxo de saída escrever bytes.

A grande vantagem dessa abstração pode ser mostrada em um método qualquer que utiliza um `OutputStream` recebido como argumento para escrever em um fluxo de saída. Para onde o método está escrevendo? Não se sabe e não importa: quando o sistema precisar escrever em um arquivo ou em uma socket basta chamar o mesmo método!

java.io.File

A classe `java.io.File` dá acesso ao programa aos recursos de arquivos do sistema operacional. É importante lembrar que instâncias dessa classe podem ser tanto arquivos como diretórios ou até mesmo não existir.

Através de objetos desse tipo é possível listar diretórios e arquivos, descobrir tamanho e características de recursos etc.

13.2 - Lendo bytes e caracteres

O primeiro exemplo de leitura costuma ser dado através do fluxo de entrada padrão, o teclado:

```
1.  import java.io.*;
2.
```

```
3.  public class TestaFluxoInputStream {
4.
5.     public static void main(String args[]) {
6.           try {
7.                 leFluxo(System.in);
8.           } catch (IOException ex) {
9.                 System.out.println(ex.getMessage());
10.          }
11.    }
12.
13.    public static void leFluxo(InputStream is) throws IOException {
14.          int proximoByte = is.read();
15.          System.out.println(proximoByte);
16.    }
17.
18. }
```

É fácil perceber que para aplicações que trabalham com caracteres fica difícil usar a classe `InputStream` diretamente. O pacote `java.io` já vem com classes que trabalham caracteres no lugar de `byte`s e elas são do tipo `Reader` para ler de uma `InputStream` e do tipo `Writer` para escrever em uma `OutputStream`:

```
1.  import java.io.*;
2.
3.  public class TestaFluxoReader {
4.
5.     public static void main(String args[]) {
6.           try {
7.                 leFluxo(System.in);
8.           } catch (IOException ex) {
9.                 System.out.println(ex.getMessage());
10.          }
11.    }
12.
13.    public static void leFluxo(InputStream is) throws IOException {
14.          InputStreamReader reader = new InputStreamReader(is);
15.          char proximoCaracter = (char) reader.read();
16.          System.out.println(proximoCaracter);
17.    }
18.
19. }
```

Wrapper

O `InputStreamReader` é um wrapper para objetos do tipo `Reader`. Ele esconde um `Reader` como atributo dentro dele e ao chamar os métodos do `InputStreamReader` ele delega a função para o reader de acordo com suas necessidades.

Essa técnica é muito utilizada em diversas bibliotecas para "adicionar" funcionalidades a uma classe de forma opcional. Ela também é a base da técnica de composição.

13.3 - Lendo Strings

Apesar da classe abstrata `Reader` já ajudar no trabalho com caracteres ainda fica difícil ler `String`s inteiras. Para trabalhar com `String`s utilizamos uma classe de ajuda chamada `BufferedReader`.

BufferedReader e BufferedWriter

As duas classes que trabalham com um buffer são muito utilizadas pois aceleram o trabalho tanto em rede através de sockets como também quando utilizando arquivos pois não força escrever e ler freqüentemente dados de e para o disco rígido.

O código a seguir lê uma linha de texto do usuário:

```
1.  import java.io.*;
2.
3.  public class TestaFluxoBufferedReader {
4.
5.     public static void main(String args[]) {
6.           try {
7.                 leFluxo(System.in);
8.           } catch (IOException ex) {
9.                 System.out.println(ex.getMessage());
10.          }
11.    }
12.
13.    public static void leFluxo(InputStream is) throws IOException {
14.          InputStreamReader reader = new InputStreamReader(is);
15.          BufferedReader buffer = new BufferedReader(reader);
16.          String linha = buffer.readLine();
17.          System.out.println(linha);
18.    }
19.
20. }
```

Uma simples alteração no exemplo anterior permite que um arquivo seja lido. Basta abrir o arquivo criando um objeto `java.io.FileInputStream` e utilizar esse objeto na chamada ao método `leFluxo`:

```
1.  import java.io.*;
2.
3.  public class TestaFluxoFileInputStream {
4.
5.     public static void main(String args[]) {
6.           try {
7.                 // abre o arquivo
8.                 FileInputStream is = new FileInputStream("arquivo.txt");
9.                 // chama o método
10.                leFluxo(is);
11.                // fecha o arquivo
12.                is.close();
13.          } catch (IOException ex) {
14.                System.out.println(ex.getMessage());
15.          }
16.    }
17.
18.    public static void leFluxo(InputStream is) throws IOException {
19.          InputStreamReader reader = new InputStreamReader(is);
20.          BufferedReader buffer = new BufferedReader(reader);
21.          String linha = buffer.readLine();
22.          System.out.println(linha);
23.    }
24.
25. }
```

Como foi dito desde o começo, para alterar a entrada de um ponto para outro basta usar objetos diferentes de entrada, todo o resto do programa continua o mesmo!

Lendo um arquivo inteiro

Para ler um arquivo inteiro precisamos checar a linha lida pelo valor null, que representa o fim dele:

```
// sempre
while(true) {

      // lê uma linha
      String linha = buffer.readLine();
```

```
        // se chegou ao final
        if(linha == null) {
              // sai do loop
              break;
        }

        // imprime a linha lida
        System.out.println(linha);

}
```

13.4 - Fluxo de saída

O fluxo de saída padrão (`System.out`) é do tipo `PrintWriter` e possui um método chamado write para escrever `String`s.

Do mesmo modo que o `InputStream`, existe o OutputStream que escreve `byte`s e foi criado uma classe de ajuda chamada `Writer` para trabalhar com caracteres enquanto a classe `OutputStreamWriter` faz a ponte entre o `OutputStream` e o `Writer`.

Por último, é possível usar o `PrintWriter` que funciona como um wrapper para escrever `String`s:

```
1.  import java.io.*;
2.
3.  public class TestaFluxoPrintWriter {
4.
5.     public static void main(String args[]) {
6.           try {
7.                 escreveFluxo(System.out);
8.           } catch (IOException ex) {
9.                 System.out.println(ex.getMessage());
10.          }
11.    }
12.
13.    public static void escreveFluxo(OutputStream os) throws IOException {
14.          PrintWriter printer = new PrintWriter(os,true);
15.          printer.println("nova linha");
16.    }
17.
18. }
```

E agora escrevendo para um arquivo:

```
1.  import java.io.*;
2.
3.  public class TestaFluxoFileOutputStream {
4.
5.     public static void main(String args[]) {
6.           try {
7.                 FileOutputStream os = new FileOutputStream("arquivo.txt");
8.                 escreveFluxo(os);
9.                 // fecha o arquivo
10.                os.close();
11.          } catch (IOException ex) {
12.                System.out.println(ex.getMessage());
13.          }
14.    }
15.
16.    public static void escreveFluxo(OutputStream os) throws IOException {
17.          PrintWriter printer = new PrintWriter(os,true);
18.          printer.println("nova linha");
19.    }
20.
```

```
21. }
```

Como no exemplo de leitura, através da classe abstrata `Writer` é possível abstrair e generalizar diversos métodos e comportamentos do nosso programa!

BufferedWriter

O uso do BufferedWriter é basicamente o mesmo que o BufferedReader, ele implementa um buffer de escrita através de um wrapper para um objeto da classe Writer.

13.5 - Um pouco mais...

1-) Procure informações sobre a class `java.io.File` e suas funcionalidades. (use a documentação do tipo javadoc)

2-) Existem duas classes chamadas `java.io.FileReader` e `java.io.FileWriter`. Elas são atalhos para a leitura e escrita de arquivos.

13.6 - Exercícios

1-) Crie um programa (simplesmente uma classe com um main) que leia da entrada padrão. Para isso você vai precisar de um `BufferedReader` que leia do `System.in` de alguma forma:

```
InputStream is = System.in;
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);

String s = // leia a primeira linha
while (/* linha é diferente de null? */) {
      // ... imprime a linha e leia a próxima do BufferedReader
}
```

2-) Troque o `System.in` por um `FileInputStream`:

```
InputStream is = new FileInputStream("arquivo.txt");
```

Seu programa agora le todas as linhas desse arquivo. Repare a utilização do polimorfismo. Como ambos são `InputStream`, isso faz com que eles se encaixem no `InputStreamReader`.

3-) Altere seu programa para que ele leia do arquivo, e em vez de jogar na tela, jogue em um arquivo. Para isso você pode usar o `BufferedWriter`:

```
OutputStream os = new FileOutputStream("saida");
OutputStreamWriter osw = new OutputStreamWriter(os);
BufferedWriter bw = new BufferedWriter(osw);
```

Agora, dentro do loop de leitura do teclado, você deve usar `bw.write(x)`, onde x é a linha que você leu. Usa `bw.newLine()` para pular de linha. Não se esqueça de no término do loop dar um `bw.close()`;

4-) Altere novamente o programa para ele virar um pequeno editor: lê do teclado e escreve em arquivo.

5-) (Opcional) No java 1.5 temos a classe java.util.Scanner que facilita bastante o trabalho de ler de um `InputStream`. Além disso, a classe `PrintStrem` possui agora um construtor que ja recebe o nome de um arquivo como argumento.

Dessa forma a leitura do teclado com saída para um arquivo ficou muito simples:

```
Scanner s = new Scanner(System.in);
PrintStream ps = new PrintStream("arquivo.txt");
while(s.hasNextLine()) {
      ps.println(s.nextLine());
}
```

Substitua o seu código antigo para utilizar essas classes. É bom entender que internamente essas classes trabalham de maneira muito parecida com a qual estavamos fazendo.

13.7 - Desafios

1-) Crie um programa que dado um diretório, lista os sub-diretórios e arquivos do mesmo.

capítulo 14

Collections framework

"A amizade é um contrato segundo o qual nos comprometemos a prestar pequenos favores para que no-los retribuam com grandes."
Baron de la Brede et de Montesquieu -

Ao término desse capítulo você será capaz de:
- utilizar arrays, lists, sets ou maps dependendo da necessidade do programa;
- iterar e ordenar listas e coleções e
- usar mapas para inserção e busca de objetos.

14.1 - Arrays

ARRAYS

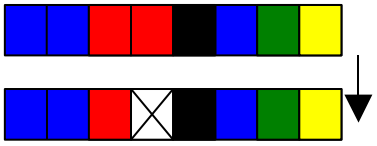
VETOR

A utilização de arrays é complicada em muitos pontos:

- não podemos redimensionar uma array em Java;

- é impossível buscar diretamente por um determinado elemento para o qual não se sabe o índice;

- não conseguimos saber quantas posições da array já foram populadas sem criar, para isso, métodos auxiliares.



Na figura acima, você pode ver um array que antes estava sendo completamente utilizado, e que depois teve um de seus elementos removidos.

Supondo que os dados armazenados representem contas, o que acontece quando precisarmos inserir uma nova conta no banco?

Precisaremos procurar por um espaço vazio?

Iremos guardar em alguma estrutura de dados externa as posições vazias?

E se não houver espaço vazio? Teríamos de criar um array maior e copiar os dados do antigo para ele?

Há mais questões: como sei quantas posições estão sendo usadas no array? Vou precisar sempre percorrer o array inteiro para conseguir essa informação?

Além dessas dificuldades que as arrays apresentavam, faltava um conjunto robusto de classes para suprir a necessidade de estruturas de dados básicas, como

listas ligadas e tabelas de espalhamento.

COLLECTIONS

Com esses e outros objetivos em mente, a Sun criou um conjunto de classes e interfaces conhecido como **Collections Framework** que reside no pacote `java.util`.

Collections

A **API** do **Collections** é robusta e possui diversas classes que representam estruturas de dados avançadas.

Por exemplo, não é necessário reinventar a roda e criar uma lista ligada mas sim utilizar aquela que a Sun disponibilizou.

14.2 - Principais interfaces

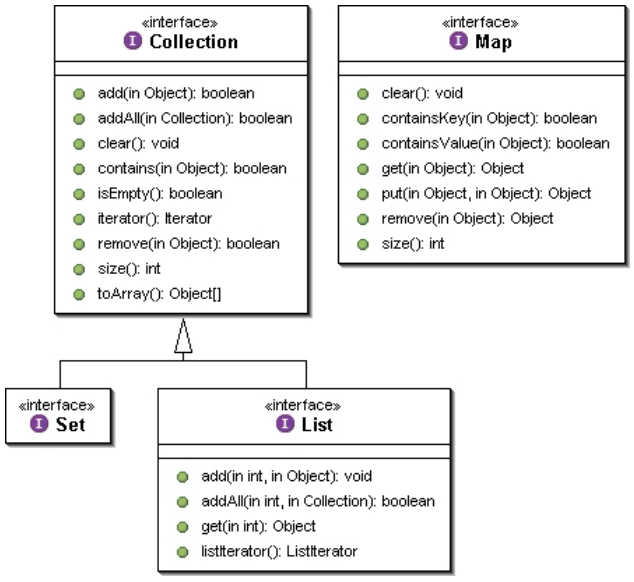
As coleções têm como base a interface `Collection`, que define métodos para adicionar e remover um elemento, verificar se ele está na coleção entre outras operações, como mostra a tabela a seguir:

boolean `add(`**Object**`)`	Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados, este método retorna `true` ou `false` indicando se a adição foi efetuada com sucesso.
boolean `remove(`**Object**`)`	Remove determinado elemento da coleção. Se ele não existia, retorna `false`.
int `size()`	Retorna a quantidade de elementos existentes na coleção.
boolean `contains(`**Object**`)`	Procura por determinado elemento na coleção, e retorna verdadeiro caso ele exista. Esta comparação é feita baseando-se no método `equals()` do objeto, e não através do operador `==`.
`Iterator iterator()`	Retorna um objeto que possibilita percorrer os elementos daquela coleção.

SET

LIST

Uma coleção pode implementar diretamente a interface `Collection`, porém existem duas subinterfaces que são amplamente utilizadas: `Set` e `List`.



A interface `Set` define um conjunto de elementos únicos enquanto a interface `List` permite a réplica de elementos.

A busca em um `Set` é mais rápida que em um objeto do tipo `List` porém a inserção é mais lenta ao comparar os algoritmos dos dois objetos.

MAP

A interface `Map` faz parte do framework mas não estende `Collection`.

Veremos detalhes sobre cada uma dessas interfaces e suas principais

implementações a seguir.

14.3 - Como ficamos no Java 5.0

A interface `Collection` agora é parametrizada. Isto é, temos uma `Collection<Tipo>` que indica qual é o **tipo** que estamos trabalhando com essa coleção. Com isso definimos o que os métodos vão receber e devolver. Uma `Collection<Tipo>` tem métodos como `contains(Tipo o)` e `add(Tipo o)`, o que nos da uma enorme segurança em tempo de compilação, evitando castings.

14.4 - Classe de exemplo

Para este capítulo iremos utilizar a classe a seguir. Usamos os atribuitos públicos apenas para demonstração, pois já vimos que esta é uma **péssima** prática.

```
1.  public class Pais {
2.
3.     // o nome do país
4.     public String nome;
5.
6.     // tamanho do terreno do país
7.     public Double kmQuadrados;
8.  }
```

14.5 - Lista

Uma lista é uma coleção que permite elementos duplicados e mantendo uma ordenação específica entre os elementos.

Em outras palavras, você tem a garantia de que, quando percorrer a lista, os elementos serão encontrados em uma ordem pré-determinada, definida na hora da inserção dos mesmos.

A implementação mais famosa da interface `List` é a `ArrayList` que trabalha com uma array interna para gerar uma lista portanto ela é mais rápida na pesquisa que sua concorrente, a `LinkedList`, que é mais rápida na inserção e remoção de itens nas pontas.

Para criar um `ArrayList` basta chamar o construtor:

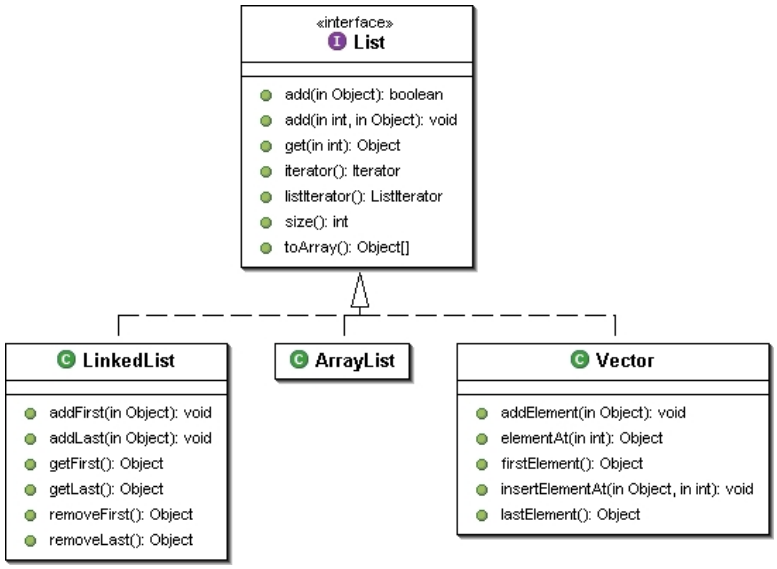
```
ArrayList lista = new ArrayList();
```

É sempre possível abstrair a lista a partir da interface `List`:

```
List mesmaLista = lista;
```

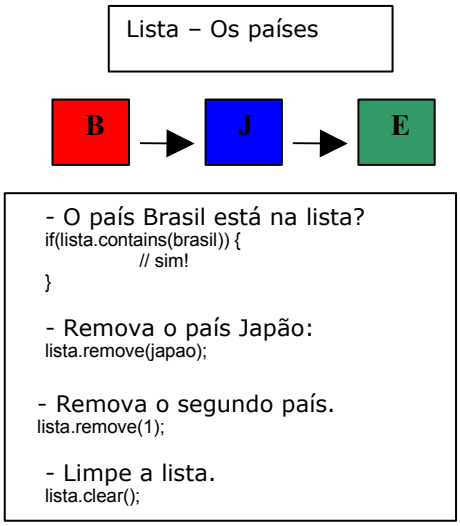
A interface `List` possui dois métodos `add`, um que recebe o objeto a ser inserido e o coloca no final da lista e um segundo que permite adicionar o elemento em qualquer posição da mesma.

A interface `List` e algumas classes que a implementam podem ser vistas no diagrama **UML** a seguir:



O exemplo a seguir gera três países e os insere no fim de uma lista.

O gráfico ao lado mostra a lista após incluir os três países Brasil, Japão e EUA.




```
1.  public class TestaArrayList {
2.
3.      public static void main(String args[]) {
4.
5.          Pais brasil = new Pais();
6.          brasil.nome = "Brasil";
7.          brasil.kmQuadrados = 8511965.0;
8.
9.          Pais japao = new Pais();
10.         japao.nome = "Japão";
11.         japao.kmQuadrados = 377815.0;
12.
13.         Pais eua = new Pais();
14.         eua.nome = "EUA";
15.         eua.kmQuadrados = 9629091.0;
16.
17.         ArrayList lista = new ArrayList();
18.         lista.add(brasil);
19.         lista.add(japao);
20.         lista.add(eua);
21.
22.         System.out.println("Tamanho :" +  lista.size());
23.     }
24. }
```

O método `get(int)` retorna o elemento na posição especificada como parâmetro, iniciando na posição zero, assim como trabalhando com uma **array** comum. Através dele e do método `size` fica fácil percorrer uma `ArrayList`.

Vale lembrar que o exemplo a seguir mostra que, em uma lista, a ordem dos elementos não é alterada:

```
1.  public class MostraArrayList {
2.
3.    public static void main(String args[]) {
4.
5.          Pais brasil = new Pais();
6.          brasil.nome = "Brasil";
7.          brasil.kmQuadrados = 8511965.0;
8.
9.          Pais japao = new Pais();
```

```
10.           japao.nome = "Japão";
11.           japao.kmQuadrados = 377815.0;
12.
13.           Pais eua = new Pais();
14.           eua.nome = "EUA";
15.           eua.kmQuadrados = 9629091.0;
16.
17.           ArrayList lista = new ArrayList();
18.           lista.add(brasil);
19.           lista.add(japao);
20.           lista.add(eua);
21.
22.           // passa por cada item, até lista.size()
23.           for(int i = 0; i != lista.size(); i++) {
24.                 // faz o cast
25.                 Pais paisAtual = (Pais) lista.get(i);
26.
27.                 // imprime o valor
28.                 System.out.println(paisAtual.nome);
29.
30.           }
31.     }
32. }
```

É importante lembrar que uma coleção sempre trabalha com `Object` portanto existe a necessidade de fazer o **cast** na referência, passando de `Object` para a classe apropriada antes de utilizar o elemento obtido da coleção.

Acesso aleatório

Algumas listas, como a `ArrayList`, tem acesso aleatório aos seus elementos: a busca por um elemento em uma determinada posição é feita de maneira imediata, sem que a lista inteira seja percorrida.

Neste caso o acesso é feito através do método `get(int)` e é muito rápido.

Uma lista é uma excelente alternativa a um array comum já que temos todos os benefícios de arrays, sem a necessidade de tomar cuidado com remoções, falta de espaço etc.

A outra implementação muito usada (`LinkedList`), fornece métodos adicionais para obter e remover o primeiro e último elemento da lista.

Vector

Outra implementação é a tradicional classe `Vector`, presente desde o Java 1.0, que foi adaptada para uso com o framework de collections, com a inclusão de novos métodos.

Ela deve ser tratada com cuidado pois lida de uma maneira diferente com processos correndo em paralelo e será mais lento que uma `ArrayList` quando não houver acesso simultâneo aos dados.

14.6 - Uma Lista no Java 5.0

A interface `List` também é parametrizada. Podemos passar um tipo para ela, indicando o tipo o qual aquela lista trabalha. As classes concretas também são parametrizadas, então podemos dar um `new`, passando o parâmetro:

```
Pais brasil = new Pais();
brasil.nome = "Brasil";
brasil.kmQuadrados = 1000;

ArrayList<Pais> lista = new ArrayList<Pais>();
```

```
lista.add(brasil);

// repare que não temos o casting!!
Pais paisAtual = lista.get(0);
```

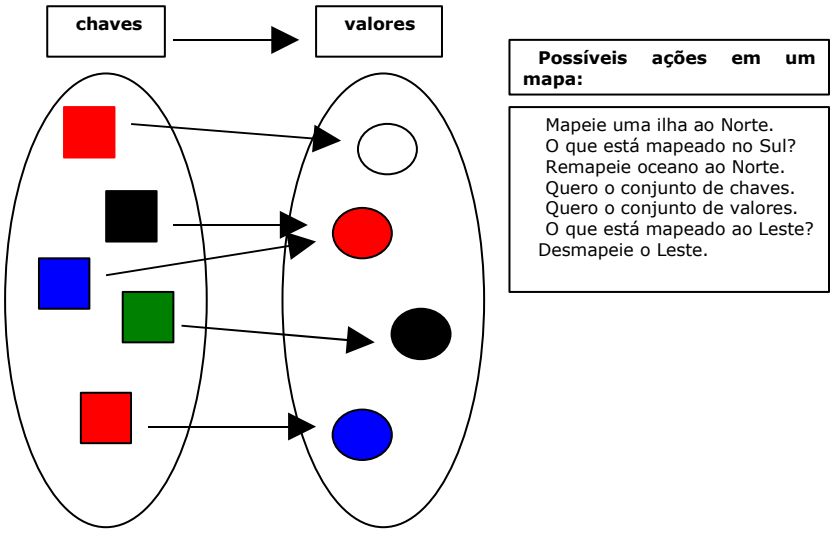
Repare a ausência de casting! Outro ponto interessante é que teríamos um erro de compilação se tentassemos passar como argumento algo não compatível com `Pais` para o método `add()`.

```
ArrayList<Pais> lista = new ArrayList<Pais>();
lista.add("string qualquer"); // não compila!
```

14.7 - Mapas

Um mapa é composto de uma associação de um objeto chave a um objeto valor.

Ele é um mapa pois é possível usá-lo para mapear uma chave, por exemplo: mapeie o valor "Estados Unidos" à chave "norte", ou mapeie "Oceano" à chave "sul".



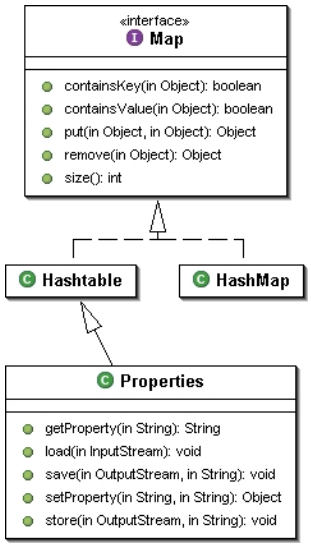
Algumas linguagens, como Perl ou PHP, possuem suporte nativo a mapas, onde são conhecidos como matrizes associativas.

O método `put(Object, Object)` da interface `Map` recebe a chave e o valor de uma nova associação. Para saber o que está associado a um determinado objeto-chave, passa-se esse objeto no método `get(Object)`. Observe o exemplo:

```
String guiana = "Guiana Francesa";
String uruguai = "Uruguai";

// cria o mapa
Map mapa = new HashMap();

// adiciona algo ao norte e algo ao sul
mapa.put("norte", guiana);
mapa.put("sul", uruguai);

// que pais esta associado a String "sul"?
Object elemento = mapa.get("sul");
String pais = (String) elemento;
```

Um mapa, assim como as coleções, trabalha diretamente com `Object`s, o que torna necessário o casting no momento que recuperar elementos. Suas principais

implementações são o `HashMap` e o `Hashtable`.



Apesar do mapa fazer parte do framework, ele não implementa a interface `Collection`, por ter um comportamento bem diferente. Porém, as coleções internas de um mapa (a de chaves e a de valores, ver Figura 7) são acessíveis por métodos definidos na interface `Map`.

O método `keySet()` retorna um `Set` com as chaves daquele mapa, e o método `values()` retorna a `Collection` com todos os valores que foram associados a alguma das chaves.

Um mapa importante é a tradicional classe `Properties`, que mapeia strings e é muito utilizada para a configuração de aplicações.

```
Properties config = new Properties();

config.setProperty("database.login", "scott");
config.setProperty("database.password", "tiger");
config.setProperty("database.url","jdbc:mysql:/localhost/teste");

// muitas linhas depois…

String login = config.getProperty("database.login");
String password = config.getProperty("database.password");
String url = config.getProperty("database.url");
DriverManager.getConnection(url, login, password);
```

Repare que não houve a necessidade do casting para `String` no momento de recuperar os objetos associados. Isto porque a classe `Properties` foi desenhada com o propósito de trabalhar com a associação entre `Strings`.

A `Properties` possui também métodos para ler e gravar o mapeamento com base em um arquivo texto, facilitando muito a sua persistência.

14.8 - Mapas no Java 5.0

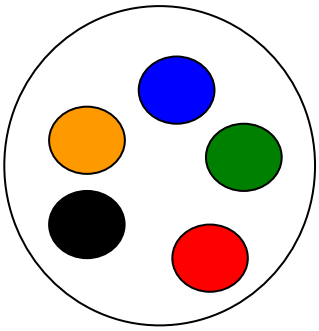
Assim como as coleções, um mapa no Java 5.0 é parametrizado. O interessante é que ele recebe dois parâmetros: a chave e o valor:

```
Pais guiana = new Pais("Guiana Francesa");
Pais uruguai = new Pais("Uruguai");

// cria o mapa, passando parametros <Chave, Valor>
Map<String,Pais> mapa = new HashMap<String,Pais>();

// adiciona algo ao norte e algo ao sul
mapa.put("norte", guiana);
mapa.put("sul", uruguai);

// que pais esta associado a String "sul"? Sem casting:
Pais pais = mapa.get("norte");
```

Aqui, como no caso da Lista, se você tentar colocar algo diferente de String -> Pais, vai ter um erro de compilação.

14.9 - Conjunto

Um conjunto (`Set`) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.

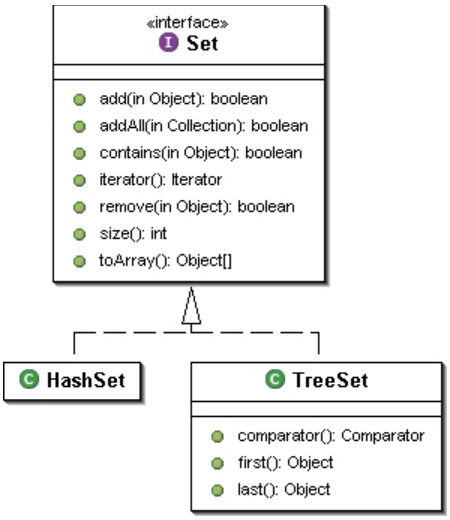


Possíveis ações em um conjunto:

- A camiseta Azul está no conjunto?
- Remova a camiseta Azul.
- Adicione a camiseta Vermelha.
- Limpe o conjunto.

- Não existem elementos duplicados!

- Ao percorrer um conjunto, sua ordem não é conhecida!



Outra característica fundamental dele é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto.

Tal ordem varia de implementação para implementação.

Um conjunto é representado pela interface `Set` e tem como suas principais implementações as classes `HashSet` e `TreeSet`.

O código a seguir cria um conjunto e adiciona três itens, apesar de tentar adicionar quatro:

```
Set conjunto = new HashSet();
conjunto.add("item 1");
conjunto.add("item 2");
conjunto.add("item 3");
conjunto.add("item 3");

// imprime a sequência na tela
System.out.println(conjunto);
```

O resultado são os elementos do conjunto, a ordem na qual eles aparecem podem ou não ser a ordem na qual eles foram inseridos e é incorreto supor que será sempre a mesma ordem!

Ordenando um set

Seria possível usar uma outra implementação de conjuntos, como um TreeSet, que insere os elementos, de tal forma, que quando forem percorridos, aparecem em uma ordem definida pelo método de comparação entre seus elementos. Esse método é definido pela interface java.lang.Comparable.

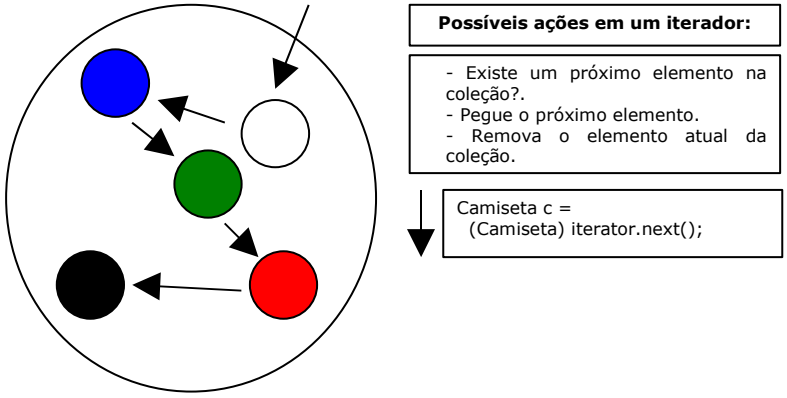
Um conjunto também pode ser parametrizado no Java 5.0, eliminando necessidade de castings.

14.10 - Iterando sobre coleções

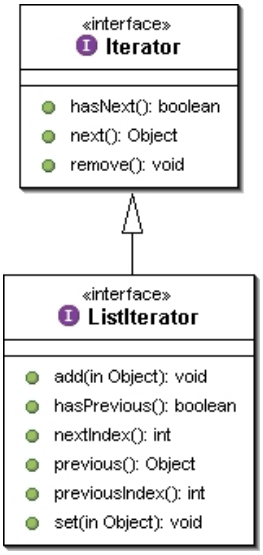
Como percorrer os elementos de uma coleção? Se for uma lista, podemos sempre utilizar um laço `for`, chamando o método `get` para cada elemento. Mas e se a coleção não permitir indexação?

Por exemplo, um `Set` não possui uma função para pegar o primeiro, o segundo ou o quinto elemento do conjunto...

Toda coleção fornece acesso a um iterator, um objeto que implementa a interface `Iterator`, que conhece internamente a coleção e dá acesso a todos os seus elementos, como a figura abaixo mostra.



Primeiro criamos um Iterator que entra na coleção.



A cada chamada do método `next`, o `Iterator` retorna o próximo objeto do conjunto.

Um iterator pode ser obtido com o método `iterator()` de `Collection`, por exemplo:

```
Iterator i = lista.iterator();
```

A interface `Iterator` possui dois métodos principais: `hasNext()` (com retorno booleano) indica se ainda existe um elemento a ser percorrido; `next()` retorna o próximo objeto.

Voltando ao exemplo do conjunto de strings, vamos percorrer o conjunto:

```
1.  // popula o conjunto
2.  Set conjunto = new HashSet();
3.  conjunto.add("item 1");
4.  conjunto.add("item 2");
5.  conjunto.add("item 3");
6.
7.  // retorna o iterator
8.  Iterator i = conjunto.iterator();
9.  while (i.hasNext()) {
10.
11.    // recebe a palavra
12.    Object elemento = i.next();
13.    String palavra = (String) elemento;
```

```
14.
15.    // mostra a palavra
16.    System.out.println(palavra);
17. }
```

O `while` anterior só termina quando todos os elementos do conjunto forem percorridos, isto é, quando o método `hasNext` mencionar que não existem mais itens.

Em que ordem serão acessados os elementos?

Numa lista, os elementos irão aparecer de acordo com o índice em que foram inseridos, isto é, de acordo com o que foi pré-determinado. Em um conjunto, a ordem depende da implementação da interface `Set`.

Por que o `Set` é então tão importante e usado?

Para perceber se um item já existe em uma lista é muito mais rápido usar um `Set` do que um `List`, e os `TreeSet`s já vem ordenados de acordo com as características que desejarmos!

ListIterator

Uma lista fornece, além de acesso a um Iterator, um ListIterator, que oferece recursos adicionais, específicos para listas.

Usando o ListIterator você pode, por exemplo, adicionar um elemento na lista ou voltar para o elemento que foi "iterado" anteriormente.

14.11 - Iterando coleções no java 5.0

Podemos utilizar a mesma sintaxe do *enhanced-for* para percorrer qualquer `Collection`.

```
1.  Set conjunto = new HashSet();
2.  conjunto.add("item 1");
3.  conjunto.add("item 2");
4.  conjunto.add("item 3");
5.
6.  // retorna o iterator
7.  for(Object elemento : conjunto) {
8.     String palavra = (String) elemento;
9.     System.out.println(palavra);
10. }
```

O Java vai usar o iterator da `Collection` dada para percorrer a coleção. Se você já estiver usando uma coleção parametrizada, o `for` pode ser feito utilizando um tipo mais específico:

```
1.  Set<String> conjunto = new HashSet<String>();
2.  conjunto.add("item 1");
3.  conjunto.add("item 2");
4.  conjunto.add("item 3");
5.
6.  // retorna o iterator
7.  for(String palavra : conjunto) {
8.     System.out.println(palavra);
9.  }
```

Isso é possível por causa que o método `iterator()` da `Collection<Tipo>` devolve um `Iterator<Tipo>`.

14.12 - Ordenação

Muitas vezes queremos percorrer a nossa coleção de maneira ordenada. Mas como definir a ordem dos objetos?

Vimos anteriormente que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens e que os conjuntos são percorridos de forma (até agora) indeterminada.

É necessário capacitar nosso programa na comparação de dois objetos da coleção. Para isso existem duas maneiras:

COMPARABLE

Uma é fazer com que os elementos da sua coleção implementem a interface `java.lang.Comparable`, que define o método `int compareTo(Object)`. Este método deve retornar **zero** se o objeto comparado for **igual** a este objeto, um número **negativo** se este objeto for **menor** que o objeto dado, e um número **positivo** se este objeto for **maior** que o objeto dado.

Para ordenar as Contas por `saldo`, basta implementar o `Comparable`:

```
1.  public class Conta implements Comparable<Conta> {
2.
3.      // ... todo o codigo anterior fica aqui
4.
5.      public int compareTo(Conta outra) {
6.
7.        if(this.saldo < outra.saldo) {
8.          return -1;
9.        }
10.
11.       if(this.saldo > outra.saldo) {
12.         return 1;
13.       }
14.
15.       return 0;
16.
17.     }
18. }
```

Com o código anterior, nossa classe tornou-se "**comparável**": dados dois objetos da classe, conseguimos dizer se um objeto é maior, menor ou igual ao outro, segundo algum critério por nós definido. No nosso caso, a comparação será feita baseando-se no saldo da conta.

Como fazemos para ordenar a coleção? Podemos usar os métodos da classe utilitária `Collections`, que inclui uma série de métodos estáticos que realizam funções comuns em coleções. Você pode ordenar uma lista, obter o maior elemento de uma coleção, entre muitas outras funcionalidades, já que seus elementos agora são comparáveis. A tabela a seguir mostra alguns dos métodos da `Collections`.

int binarySearch (**List**, **Object**)	Realiza uma busca binária por determinado elemento na lista ordenada, e retorna sua posição, ou um número negativo caso não encontrado.
Object max(**Collection**)	Retorna o maior elemento da coleção.
Object min(**Collection**)	Retorna o menor elemento da coleção.
void sort(**List**)	Ordena a lista dada em ordem ascendente.

Muitos métodos exigem que os objetos sejam comparáveis. O seguinte exemplo mostra como é simples ordenar uma lista:

```
1.  List contas = new ArrayList();
2.
3.  // adiciona a primeira conta
4.  Conta c1 = new Conta();
5.  c1.setSaldo(200);
6.  contas.add(c1);
7.
8.  // adiciona a segunda conta
9.  Conta c2 = new Conta();
10. c2.setSaldo(100);
11. contas.add(c2);
12.
13. // ordena
14. Collections.sort(contas);
```

Outro jeito de ordenar, que ordena a cada inserção, é utilizando o `TreeSet`. Ao final do código a seguir, as contas já estarão ordenadas!

```
1.  Collection contas = new TreeSet();
2.
15. // adiciona a primeira conta
16. Conta c1 = new Conta();
17. c1.setSaldo(200);
18. contas.add(c1);
19.
20. // adiciona a segunda conta
21. Conta c2 = new Conta();
22. c2.setSaldo(100);
23. contas.add(c2);
```

Opcionalmente, em vez de implementar `Comparable`, você pode fornecer um `java.util.Comparator` como argumento a alguns dos métodos da `Collections`. Por exemplo, temos também o método `sort(List, Comparator)`, que é muito utilizado se você não tem acesso ao código fonte da classe dos seus elementos, ou se for necessário ordenar uma lista de várias maneiras diferentes.

java.util.Collections no java 5.0

Os métodos estáticos das classes utilitárias do `java.util` são parametrizados. Por exemplo, dada uma `List<String>` para a `binarySearch`, você já recebe uma `String`, sem necessidade de casting.

Jakarta Commons Collections

O projeto **Jakarta** possui uma série de **APIs** menores, conhecidas como **jakarta-commons**. Dentre elas, existe algumas classes de coleções diferentes das encontradas no `java.util`, assim como algumas classes auxiliares.

As coleções do jakarta incluem a interessante interface Predicate, onde você pode definir uma "**query**" para executar sobre uma coleção, trazendo um `Iterator` que itera apenas sobre os objetos que obedecem ao predicado dado.

Por exemplo, com uma coleção de carros, é necessário um iterator apenas para os carros verdes.

Entre outras, as **Jakarta Collections** possuem coleções para tipos primitivos, eliminando a necessidade de utilizar as **classes wrapper** do Java. Também existe um heap binário, no qual a coleção está ordenada de tal maneira que é possível obter o valor máximo (ou mínimo) rapidamente, em tempo constante (ou O(1)).

Equals e HashCode

Muitas das coleções do java guardam os objetos dentro de tabelas de hash. Essas tabelas são utilizadas para que a pesquisa de um objeto seja feira de maneira rápida.

Como funciona? Cada objeto é "classificado" pelo seu `hashCode`, e com isso conseguimos espalhar cada objeto agrupando pelo `hashCode`. Quando vou buscar determinado objeto, só vamos procurar entre os elementos que estão no grupo daquele hashCode. Dentro desse grupo vamos testando o objeto procurado com o candidato usando `equals()`.

Para que isso funcione direito, o método `hashCode` de cada objeto deve retornar o mesmo valor para dois objetos se eles são considerados `equals`. Em outras palavras:

a.equals(b) implica a.hashCode() == b.hashCode()

Implementar `hashCode` de tal maneira que ele retorne valores diferentes para dois objetos considerados `equals` quebra o contrato de `Object`, e resultará em collections não achando objetos iguais dentro de uma mesma coleção.

Boas práticas

As coleções do Java oferecem grande flexibilidade ao usuário. A perda de performance em relação a utilização de arrays é irrelevante, mas deve-se tomar algumas precauções:

- Grande parte das coleções usam internamente uma array para armazenar os seus dados. Quando essa array não é mais suficiente, é criada uma maior e o conteúdo da antiga é copiado. Este processo pode acontecer muitas vezes no caso de você ter uma coleção que cresce muito. Você deve então criar uma coleção já com uma capacidade grande, para evitar o excesso de redimensionamento.

- Evite usar coleções que guardam os elementos pela sua ordem de comparação quando não há necessidade. Um `TreeSet` gasta computacionalmente **O(log(n))** para inserir (ele utiliza uma árvore rubro-negra como implementação), enquanto o `HashSet` gasta apenas **O(1)**.

– Não itere sobre uma `List` utilizando um for de 0 até `list.size()`, e usando `get(int)` para receber os objetos. Enquanto isso parece atraente, algumas implementações da `List` não são de acesso aleatório como a `LinkedList`, fazendo esse código ter uma péssima performance computacional. (use `Iterator`)

14.13 - Exercícios

1-) Crie um código que insira 100 mil números numa `ArrayList`. Agora faça um for que procure por todos os elementos usando o contains. Troque a `ArrayList` por um `HashSet` e verifique o tempo que vai demorar.

Repare que se você declarar a coleção e der new assim:

```
Collection<Integer> teste = new ArrayList<Integer>();
```

em vez de

```
ArrayList<Integer> teste = new ArrayList<Integer>();
```

É garantido que vai ter de alterar só essa linha para substituir a

implementação por `HashSet`. Estamos aqui usando o polimorfismo para nos proteger que mudanças de implementação venham nos obrigar a alterar muito código.

Esse é um código extremamente elegante e flexível. Obviamente algumas vezes não conseguimos trabalhar dessa forma, e precisamos usar uma interface mais específica ou mesmo nos referir ao objeto pela sua implementação para poder chamar métodos mais específicos (por exemplo, `TreeSet` tem mais métodos que em `Set`, assim como `LinkedList` em `List`).

2-) Faça sua classe `Conta` implementar a interface `Comparable<Conta>`. Utilize o critério que desejar, um exemplo é ordenar pelo número da conta ou pelo seu saldo (como visto no código deste capítulo).

Crie diversas instâncias de contas e adicione-as numa `List<Conta>`. Use o `Collections.sort()` nessa lista:

```
List<Conta> contas = new ArrayList<Conta>();

Conta c1 = new Conta();
c1.deposita(150);
contas.add(c1);

Conta c2 = new Conta();
c2.deposita(100);
contas.add(c2);

Conta c3 = new Conta();
c3.deposita(200);
contas.add(c3);

Collections.sort(contas);
```

Se preferir insira novas contas através de um laço.

Imprima a referência para essa lista (`System.out.println(contas)`). Repare que o `toString` de uma Array/LinkedList é reescrito.

3-) O que teria acontecido se a classe `Conta` não implementasse `Comparable<Conta>`?

4-) (Opcional) Crie uma classe `Banco` que possui uma array de `Conta`. Repare que numa array de `Conta` você pode colocar tanto `ContaConrrente` quanto `ContaPoupanca`. Crie um metodo `void adiciona(Conta c)`, um método `Conta pega(int x)` e outro `int pegaTotalDeContas()`, muito similar a relação anterior de Empresa-Funcionario.

5-) (opcional) Crie um método na classe `Banco` que busca por um `Funcionario`.

6-) (opcional) Crie o método `hashCode` para a sua conta, de forma que ele respeito o equals de que duas contas sao equals quando tem o mesmo numero. Verifique se sua classe funciona corretamente num `HashSet`. Remova o método `hashCode`. Continua funcionando?

Dominar o uso e o funcionamento do `hashCode` é fundamental para o bom programador.

14.14 - Desafios

1-) Gere todos os números entre 1 e 1000 e ordene em ordem não crescente

utilizando um `TreeSet`.

2-) Gere todos os números entre 1 e 1000 e ordene em ordem não crescente utilizando um `ArrayList`.

capítulo 15

Threads

"O único lugar onde o sucesso vem antes do trabalho é no dicionário."
Albert Einstein -

Ao término desse capítulo, você será capaz de:
- executar tarefas simultaneamente;
- colocar tarefas para aguardar um sinal;
- esperar alguns segundos para continuar a execução de um programa.

15.1 - Linhas de execução

EXECUÇÃO CONCORRENTE

Podemos, em Java, facilmente criar uma classe que tem um método que vai ser executado concorrentemente com o seu `main`. Para isso, você precisa implementar a interface `Runnable`, que define o método `run`.

RUNNABLE

```
1.  package br.com.caelum.threads;
2.
3.  class Programa implements Runnable {
4.
5.     private int id;
6.
7.     //colocar getter e setter pro atributo id
8.
9.     public void run() {
10.          for (int i = 0; i < 10000; i++) {
11.                System.out.println("Programa " + id + " valor: " + i);
12.          }
13.    }
14. }
```

THREAD

Para você criar essa nova linha de execução (`Thread`), é muito simples.

```
1.  package br.com.caelum.threads;
2.
3.  class Teste {
4.     public static void main(String[] args) {
5.           Programa p = new Programa();
6.           p.setId(1);
7.           Thread t = new Thread(p);
8.           t.start();
9.     }
10. }
```

Mas, e se tivermos o seguinte caso:

```
1.  package br.com.caelum.threads;
2.
3.  class Teste {
4.     public static void main(String[] args) {
5.           Programa p1 = new Programa();
6.           p1.setId(1);
7.           Programa p2 = new Programa();
8.           p2.setId(2);
```

```
9.
10.           Thread t1 = new Thread(p1);
11.           Thread t2 = new Thread(p2);
12.
13.           t1.start();
14.           t2.start();
15.     }
16. }
```

O que vai aparecer? Duas vezes de 0 a 9999? Intercalado 0,0,1,1 ....? Você não sabe!

Você não tem o controle, e nem sabe, quem executa em cada momento. A idéia de criar uma `Thread` é exatamente estar pedindo que as execuções desses dois códigos seja concorrente, isto é, não importa a ordem para você. Se a ordem de execução importasse, você deveria ter colocado tudo numa única linha de execução.

Quando você chama a **virtual machine**, ela cria uma `Thread` para rodar o seu `main`. E você pode estar criando (disparando) outras.

Dormindo

SLEEP

Para que a thread atual durma basta chamar o método a seguir, por exemplo, para dormir 3 segundos:

```
Thread.sleep(3 * 1000);
```

15.2 - Criando uma subclasse da classe Thread

A classe Thread implementa Runnable. Então, você pode criar uma subclasse dela e reescrever o run, que na classe Thread não faz nada.

```
class MinhaThread extends Thread{
      public void run() {
            // código a ser executado pela Thread
      }
}
```

E no seu `main`:

```
MinhaThread t = new MinhaThread();
t.start();
```

Apesar de ser um código mais simples, você está usando herança apenas por facilidade, e não por polimorfismo, que seria a grande vantagem. Prefira implementar `Runnable` a herdar de `Thread`.

15.3 - Garbage Collector

GARBAGE COLLECTOR

O **Garbage Collector** (coletor de lixo, lixeiro) é uma `Thread` responsável por jogar fora todos os objetos que não estão sendo referenciados por nenhuma outra `Thread`, seja direta ou indiretamente!

```
Conta conta1 = new Conta();
Conta conta2 = new Conta()
```

Até este momento, sabemos que temos 2 objetos em memória. E se fizermos o seguinte:

```
conta2 = conta1;
```

Quantos objetos temos em memória? Perdemos uma das referência para um dos carros que foram criados. Esse objeto já não é mais acessível. Temos então apenas um objeto em memória?

Você não sabe! Como o Garbage Collector é uma `Thread`, você não tem garantia de quando ele vai rodar. Você só sabe que algum dia aquela memória vai ser liberada.

System.gc();

Você não consegue nunca forçar que o Garbage Collector rode, mas chamando o método estático `gc` da classe `System`, você esta sugerindo para que a Virtual Machine rode o Garbage Collector naquele momento. Se sua sugestão vai ser aceita ou não, isto depende e você não tem garantias. Evite o uso deste método.

Finalizer

A classe `Object` define também um método `finalize`, que você pode reescrever. Esse método será chamado no instante antes do Garbage Collector coletar este objeto. **Não é um destrutor! Ele nem sempre será chamado!**

15.4 - Compartilhando objetos entre Threads

O uso de `Threads` começa a ficar interessante e complicado quando precisamos compartilhar objetos.

Imagine que nosso Banco iniciou uma promoção nacional onde novos clientes ganham um brinde especial ao abrir uma conta. A matriz do Banco mantém um Deposito para os brindes. Cada Agência, conforme for abrindo novas contas, retira um brinde nesse depósito.

Nosso banco possui muitas agências que vão abrindo contas diariamente e, durante a promoção, distribuindo brindes. Cada `Agencia` abre contas por si só, assim como a matriz coloca os brindes ainda disponíveis no `Deposito` independente do funcionamento de uma `Agencia`.

```
1.  package br.com.caelum.threads;
2.
3.  public class Brinde {
4.     private int id;
5.     private static int contador = 0;
6.
7.     Brinde() {
8.           this.id = contador;
9.     }
10.
11.    public String toString() {
12.          return "Brinde " + this.id;
13.    }
14. }
```

```
1.  package br.com.caelum.threads;
2.
3.  public class Deposito {
4.     private Brinde[] brindes = new Brinde[100];
5.     private int proximoBrinde = 0;
6.
7.     public void adicionaBrindeFabricado(Brinde brinde) {
8.           if (this.proximoBrinde == 100) {
9.                 throw new IllegalStateException("Não há mais espaço!");
10.          }
11.          this.brindes[this.proximoBrinde] = brinde;
```

```
12.           this.proximoBrinde++;
13.    }
14.
15.    public Brinde distribuiBrinde() {
16.           if (this.proximoBrinde == 0) {
17.                  throw new IllegalStateException("Não há mais brindes!");
18.           }
19.           this.proximoBrinde--;
20.           return this.brindes[this.proximoBrinde];
21.    }
22. }
```

IllegalStateException

Esta é uma `Exception` do pacote `java.lang`, utilizado para informar que o estado de um objeto não está legal para determinada operação. Como não há um bloco de `try`/`catch`, e nem a clausula throws, podemos deduzir que esta `Exception` é do tipo `RuntimeException`, isto é, das que não são checadas (unchecked).

É uma boa prática reaproveitar as `Exception`s existentes nos pacotes do Java em vez de sempre criar uma nova.

Temos também a nossa `Matriz` e a `Agencia`. Cada uma delas implementa `Runnable` pois cada instância será rodada como uma `Thread`, e elas sempre precisam de uma referencia a um `Deposito`, para saber onde colocar e tirar seus brindes.

```
1.  package br.com.caelum.threads;
2.
3.  public class Matriz implements Runnable {
4.     private Deposito deposito;
5.
6.     public Matriz(Deposito deposito) {
7.           this.deposito = deposito;
8.     }
9.
10.    public void run()  {
11.          for (int i = 0; i < 50; i++) {
12.                this.deposito.adicionaBrindeFabricado(new Brinde());
13.          }
14.    }
15. }
```

```
1.  package br.com.caelum.threads;
2.
3.  public class Agencia implements Runnable {
4.     private Deposito deposito;
16.
17.    public Agencia (Deposito deposito) {
18.          this.deposito = deposito;
5.     }
6.
7.     public void run()  {
8.           for (int i = 0; i < 5; i++) {
9.                 Brinde brinde = this.deposito.distribuiBrinde();
10.                System.out.println("Brinde Distribuído: " + brinde);
11.          }
12.    }
13. }
```

E dentro de uma classe de Teste, executamos nosso sistema:

```
Deposito deposito = new Deposito();

Matriz matriz = new Matriz(deposito);
Thread threadDaMatriz = new Thread(matriz);
```

```
threadDaMatriz.start();

Agencia ag1 = new Agencia(deposito);
Thread t1 = new Thread(ag1);
t1.start();

Agencia ag2 = new Agencia(deposito);
Thread t2 = new Thread(ag2);
t2.start();
```

Após todas estas `Thread`s serem iniciadas, qual é o nosso resultado? Se alguma `Agencia` tentar tirar um brinde sem ter brinde algum, uma `Exception` será disparada e aceitamos isso. Mas será esse o nosso único problema?

Pode acontecer que a `Matriz` seja colocada de lado durante o método `adicionaBrindeFabricado`, fazendo com que ele não atualize o `proximoBrinde`. Quando uma thread `Agencia` pegar um brinde, vai pegar o anterior, e no momento em que a `Matriz` voltar, colocará o novo `Brinde` num lugar errado.

15.5 - Usando um lock

O problema todo foi gerado pois as linhas dos métodos `distribuiBrinde` e `adicionaBrindeFabricado` não estão bem "**amarradas**", elas não estão sendo executadas de maneira **atômica**. Uma `Thread` pode ser colocada de lado enquanto estava no meio de um desses métodos, e uma outra pode entrar nesse método, sem que a primeira tenha concluído seu trabalho.

Uma idéia seria criar uma trava, e no momento em que uma `Thread` entrasse em um desses métodos, trancasse com uma chave a entrada para eles, dessa maneira, mesmo que sendo colocada de lado, nenhuma outra `Thread` poderia entrar nesses métodos, pois a chave estaria com a outra `Thread`.

LOCK

Podemos fazer isso em Java. Podemos usar qualquer objeto como um **lock** (trava, chave), para poder estar sincronizando em cima desse objeto, isto é, se uma `Thread` entrar em um bloco que foi definido como sincronizado por esse lock, apenas uma `Thread` poderá estar lá dentro ao mesmo tempo, pois a chave estará com ela.

SYNCHRONIZED

A palavra chave `synchronized` dá essa característica a um bloco de código, e recebe qual é o objeto que será usado como chave. A chave só é devolvida no momento em que a `Thread` que tinha essa chave sair do bloco, seja por `return` ou disparo de uma exceção.

```
23. package br.com.caelum.threads;
24.
25. public class Deposito {
26.    private Brinde[] brindes = new Brinde[100];
27.    private int proximoBrinde = 0;
28.
29.    public void adicionaBrindeFabricado(Brinde brinde) {
30.          synchronized (this) {
31.                if (this.proximoBrinde == 100) {
32.                      throw new IllegalStateException("Não há mais espaço!");
33.                }
34.                this.brindes[this.proximoBrinde] = brinde;
35.                this.proximoBrinde++;
36.          }
37.    }
38.
39.    public Brinde distribuiBrinde() {
40.          synchronized (this) {
41.                if (this.proximoBrinde == 0) {
42.                      throw new IllegalStateException("Não há mais brindes!");
```

```
43.                 }
44.                 this.proximoBrinde--;
45.                 return this.brindes[this.proximoBrinde];
46.           }
47.     }
48. }
```

Esses métodos agora são **mutuamente exclusivos**, e só executam de maneira **atômica**. `Threads` tentando pegar um lock que já está pego, ficaram em um conjunto especial esperando pela liberação do lock.

Sincronizando o bloco inteiro

É comum sempre sincronizarmos um método inteiro, e normalmente em cima do `this`.

```
public void metodo() {
      synchronized (this) {
            // conteudo do metodo
      }
}
```

Para isto, existe uma sintaxe mais simples:

```
public synchronized void metodo() {
      // conteudo do metodo
}
```

Detalhe: se o método for estático, será sincronizado usando o lock do objeto da classes (NomeDaClasse.class).

15.6 - Um pouco mais...

1-) Você pode mudar a prioridade de cada uma de suas `Thread`s, mas isto também é apenas uma sugestão.

2-) Existe um método `stop` nas `Thread`s, porque não é boa prática chamá-lo?

15.7 - Exercícios

1-) Gere diversas threads simultâneas contando de 1 até 1000. Veja o interleaving (entrelaçamento dos processos).

2-) Tente montar uma thread que conta de zero até mil e depois volta até zero infinitas vezes. Crie uma segunda Thread que deverá desligar a primeira thread após no mínimo 5 segundos.

Dica: use `Thread.sleep()` para 'dormir' no mínimo um tempo determinado.

capítulo 16

E agora?

"A primeira coisa a entender é que você não entende."
Soren Aabye Kierkegaard -

Onde continuar ao terminar o 'Java e Orientação a Objetos'.

16.1 - Exercício prático

A melhor maneira para fixar tudo o que foi visto nos capítulos anteriores é planejar e montar pequenos sistemas.

Um exercício simples é terminar o sistema do Pet Shop permitindo aos usuários comprarem itens a partir da linha de comando.

Outro sistema que pode ser atualizado é o de controle de trens, cidades e malha rodoviária.

16.2 - Certificação

Entrar em detalhes nos assuntos contidos até agora iriam no mínimo tornar cada capítulo quatro vezes maior do que já é.

Os tópicos abordados (com a adição e remoção de alguns) constituem boa parte do que é cobrado na certificação oficial para programadores da Sun.

Para maiores informações sobre certificações consulte a própria Sun, o javaranch.com ou o guj.com.br que possui diversas informações sobre o assunto. A Caelum oferece um curso de preparação para a prova de certificação como programador em Java da Sun.

16.3 - Web

Um dos principais focos de Java hoje em dia é onde a maior parte das vagas existem: programando para a web.

Um curso de servlets e jsp basta, enquanto ir adiante e ver Design Patterns, XML, acesso controlado a banco de dados e outros torna um aprendiz em mestre.

16.4 - J2EE

Após focar o conhecimento e treinar tudo o que foi aprendido pode ser uma boa idéia partir para o padrão J2EE... J2EE usa tudo que vimos aqui, e é apenas um grande conjunto de especificações.

16.5 - Frameworks

Diversos frameworks foram desenvolvidos para facilitar o trabalho de equipes

de desenvolvimento.

Aqueles que pretendem trabalhar com Java devem a qualquer custo analisar as vantagens e desvantagens da maior parte desses frameworks que diminuem o número de linha de código necessárias e facilitam o controle e organização de uma aplicação.

Por exemplo, o vRaptor é um exemplo de controlador simples e bom para iniciantes. O Hibernate é um ótimo passo, assim como o prevayler, para persistência/prevalência de objetos.

16.6 - Revistas

Diversas revistas, no Brasil e no exterior, estudam o mundo java como ninguém e podem ajudar o iniciante a conhecer muito do que está acontecendo lá fora nas aplicações comerciais.

16.7 - Grupo de Usuários

Diversos programadores com o mínimo ou máximo de conhecimento se reúnem online para a troca de dúvidas, informações e idéias sobre projetos, bibliotecas e muito mais. Um dos mais importantes e famosos no Brasil é o GUJ – www.guj.com.br

16.8 - Falando em Java

O 'Falando em Java' não para por aqui, continua com o curso de Java Distribuído incluindo web, sockets, rmi, ejb, jms e muito mais...

Consulte o site oficial do 'FJ' em www.caelum.com.br para receber mais informações.

Os autores dessa edição, Paulo Eduardo Azevedo Silveira e Guilherme de Azevedo Silveira agradecem ao leitor pelo tempo investido e esperam ter ajudado a converter mais alguém para o mundo da orientação a objetos.

capítulo 17

Apêndice A - Sockets

"Olho por olho, e o mundo acabará cego."
Mohandas Gandhi -

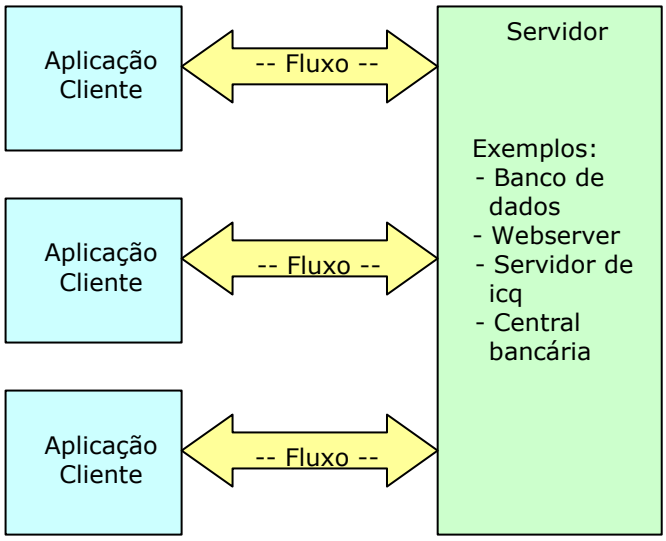
Conectando-se a máquinas remotas.

17.1 - Protocolo

TCP

Da necessidade de dois computadores se comunicarem, surgiram diversos protocolos que permitissem tal troca de informação: o protocolo que iremos estudar aqui é o **TCP** (Transmission Control Protocol).

Através do **TCP** é possível criar um **fluxo** entre dois computadores como é mostrado no diagrama abaixo:



É possível conectar mais de um cliente ao mesmo servidor, como é o caso de diversos banco de dados, webservers etc.

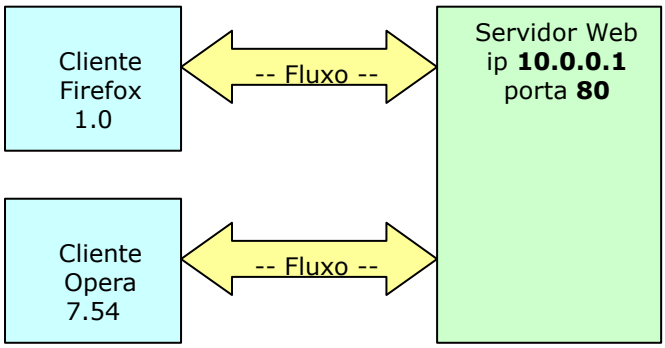
Ao escrever um programa em Java que se comunique com outra aplicação, não é necessário se preocupar com um nível tão baixo quanto o protocolo. As classes que trabalham com eles já foram disponibilizadas para serem usadas por nós no pacote `java.net`.

17.2 - Porta

Acabamos de mencionar que diversos computadores podem se conectar a um

só, mas na realidade é muito comum encontrar máquinas clientes com uma só conexão física. Então como é possível se conectar a dois pontos? Como é possível ser conectado por diversos pontos?

Todos as aplicações que estão enviando e recebendo dados fazem isso através da mesma conexão física mas o computador consegue discernir durante a chegada de novos dados quais informações pertencem a qual aplicação, mas como?



PORTA

Assim como existe o **IP** para indentificar uma máquina, a **porta** é a solução para indentificar diversas aplicações em uma máquina. Esta porta é um número de 2 bytes, **varia de 0 a 65535**. Se todas as portas de uma máquina estiverem ocupadas não é possível se conectar a ela enquanto nenhuma for liberada.

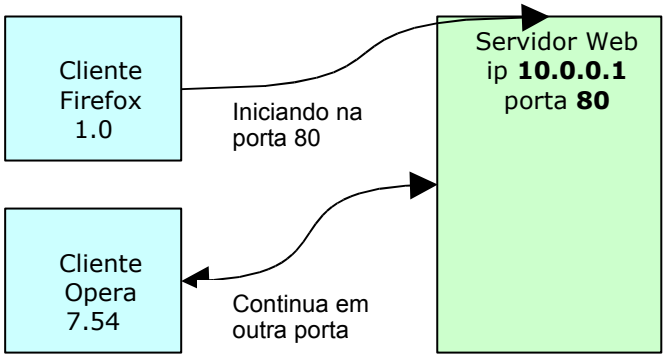
Ao configurar um servidor para rodar na porta 80 (padrão http), é possível se conectar a esse servidor através dessa porta, que junto com o ip vai formar o endereço da aplicação. Por exemplo, o servidor web da caelum.com.br pode ser representado por:

caelum.com.br:80

17.3 - Socket

Mas se um cliente se conecta a um programa rodando na porta 80 de um servidor, enquanto ele não se desconectar dessa porta será impossível que outra pessoa se conecte?

Acontece que ao efetuar a conexão, ao aceitar a conexão, o servidor redireciona o cliente de uma porta para outra, liberando novamente sua porta inicial e permitindo que outros clientes se conectem novamente.



Em Java, isso deve ser feito através de threads e o processo de aceitar a conexão deve ser rodado o mais rápido possível.

17.4 - Servidor

Iniciando agora um modelo de servidor de chat, o serviço do computador que funciona como base deve primeiro abrir uma porta e ficar ouvindo até alguém tentar

se conectar.

```
1.  import java.net.*;
2.
3.  public class Servidor {
4.
5.     public static void main(String args[]) {
6.
7.           try {
8.                 ServerSocket servidor = new ServerSocket(18981);
9.                 System.out.println("Porta 18981 aberta!");
10.                // a continuação do servidor deve ser escrita aqui
11.          } catch(IOException e) {
12.                System.out.println("Ocorreu um erro na conexão");
13.                e.printStackTrace();
14.
15.          }
16.
17.    }
18. }
```

Se o objeto for realmente criado significa que a porta 18981 estava fechada e foi aberta. Se outro programa possui o controle desta porta neste instante, é normal que o nosso exemplo não funcione pois ele não consegue utilizar uma porta que já está em uso.

Após abrir a porta, precisamos esperar por um cliente através do método `accept` da `ServerSocket`. Assim que um cliente se conectar o programa irá continuar.

```
Socket cliente = servidor.accept();
System.out.println("Nova conexão com o cliente " +
      cliente.getInetAddress().getHostAddress()
);
```

Por fim, basta ler todas as informações que o cliente nos enviar:

```
BufferedReader in = new BufferedReader(
      new InputStreamReader(cliente.getInputStream())
);

while (true) {

      String linha = in.readLine();
      if (linha == null) {
            break;
      }

      System.out.println(linha);

}
```

Agora fechamos as conexões, começando pelo fluxo:

```
in.close();
cliente.close();
servidor.close();
```

O resultado é a classe a seguir:

```
1.  import java.net.*;
2.
3.  public class Servidor {
4.
5.     public static void main(String args[]) {
6.           try {
```

```
7.                  // cria um servidor
8.                  ServerSocket servidor = new ServerSocket(18981);
9.                  System.out.println("Porta 18981 aberta!");
10.
11.                 // aceita uma conexão
12.                 Socket cliente = servidor.accept();
13.                 System.out.println("Nova conexão com o cliente " +
14.                       cliente.getInetAddress().getHostAddress()
15.                 );
16.
17.                 // cria o buffer de leitura
18.                 BufferedReader in = new BufferedReader(
19.                       new InputStreamReader(cliente.getInputStream())
20.                 );
21.
22.                 // lê ate o fim
23.                 while(true) {
24.                       String linha = in.readLine();
25.                       if (linha == null) {
26.                             break;
27.                       }
28.                       System.out.println(linha);
29.                 }
30.
31.                 // fecha tudo
32.                 in.close();
33.                 cliente.close();
34.                 servidor.close();
35.
36.           } catch (IOException e) {
37.
38.                 // em caso de erro
39.                 System.out.println("Ocorreu um erro na conexão");
40.                 e.printStackTrace();
41.           }
42.    }
43. }
```

O término

Ao fechar a primeira conexão, o servidor escrito acima encerra as operações e termina. Para receber uma nova conexão após o término da primeira basta criar um loop que começa ao receber uma nova conexão e termina quando a mesma é fechada.

Este loop não irá implementar o recurso de tratamento de diversos clientes ao mesmo tempo!

17.5 - Cliente

Agora a nossa tarefa é criar um programa cliente que envie mensagens para o servidor... o cliente é ainda mais simples que o servidor.

O código a seguir é a parte principal e tenta se conectar a um servidor no ip 127.0.0.1 (máquina local) e porta 18981.

O primeiro passo que é abrir a porta e preparar para ler os dados do cliente pode ser feito através do programa a seguir:

Cliente de Chat

- Conecta
- Lê e escreve
- Fecha a conexão

```
1.  import java.net.*;
2.
3.  public class Cliente {
4.
5.     public static void main(String args[]) {
6.
7.           try {
8.                 // conecta ao servidor
```

```
9.                  Socket cliente = new Socket("127.0.0.1",18981);
10.                 System.out.println("O cliente se conectou ao servidor!");
11.
12.                 // prepara para a leitura da linha de comando
13.                 BufferedReader in = new BufferedReader(
14.                       new InputStreamReader(System.in)
15.                 );
16.
17.                 /* inserir o resto do programa aqui */
18.
19.                 // fecha tudo
20.                 cliente.close();
21.
22.           } catch (Exception e) {
23.
24.                 // em caso de erro
25.                 System.out.println("Ocorreu um erro na conexão");
26.                 e.printStackTrace();
27.
28.           }
29.     }
30. }
```

Agora basta ler as linhas que o usuário digitar através do buffer de entrada (`in`) e jogá-las no buffer de saída:

```
PrintWriter out = new PrintWriter(cliente.getOutputStream, true);
while (true) {
      String linha = in.readLine();
      out.println(linha);
}
out.close();
```

Para testar o sistema, precisamos rodar primeiro o servidor e logo depois o cliente. Tudo o que for digitado no cliente será enviado para o servidor.

Multithreading

Para que o servidor seja capaz de trabalhar com dois clientes ao mesmo tempo é necessário criar uma thread logo após executar o método accept.

A thread criada será responsável pelo tratamento dessa conexão, enquanto o loop do servidor irá disponibilizar a porta para uma nova conexão:

```
while (true) {

      Socket cliente = servidor.accept();

      // cria um objeto que irá tratar a conexão
      TratamentoClass tratamento = new TratamentoClass(cliente);

      // cria a thread em cima deste objeto
      Thread t = new Thread(tratamento);

      // inicia a thread
      t.start();

}
```

17.6 - Exercícios

1-) Implemente o multithreading no servidor de chat

2-) Construa um sistema pequeno de servidor e cliente onde o servidor recebe cláusulas SQL e retorna para o cliente uma informação de quantas linhas foram

alteradas (insert, update, delete, create e alter) ou quantas linhas foram encontradas (select).

3-) Altere o sistema para não receber linha por linha mas sim caracter a caracter.

4-) Envie mensagens do servidor para o cliente cada vez que o cliente enviar algo ao servidor.

17.7 - Desafios

1-) Altere o sistema para criar um servidor e clientes de chat de verdade, que mostre as mensagens em todos os clientes. Dica: você pode utilizar uma lista de usuários conectados ao sistema, guardando seus OutputStreams.

17.8 - Solução

Uma s

Repare que a solução não está nem um pouco elegante: o main já faz tudo, além de não tratarmos as exceptions. O código visa apenas a mostrar o uso de uma API. É uma péssima prática colocar toda a funcionalidade do seu programa no main e também de jogar exeções para trás.

Nesta listagem faltam os devidos **imports**.

```
1.  public class Cliente {
2.
3.     public static void main(String args[]) throws Exception {
4.
5.           Socket cliente = new Socket("127.0.0.1",18981);
6.           System.out.println("O cliente se conectou ao servidor!");
7.
8.           Scanner teclado = new Scanner(System.in);
9.
10.
11.
12.                /* inserir o resto do programa aqui */
13.
14.                // fecha tudo
15.                cliente.close();
16.
17.          } catch (Exception e) {
18.
19.                // em caso de erro
20.                System.out.println("Ocorreu um erro na conexão");
21.                e.printStackTrace();
22.
23.          }
24.    }
25. }
```

Termos importantes