

PROCESSO DE DESENVOLVIMENTO JAVA WEB



INSTRUTOR: TIAGO DA ROSA VALÉRIO

E-MAIL: tiago_valerio_betha@hotmail.com

JAVA E ORIENTAÇÃO A OBJETOS

1 - JAVA

Entender um pouco da história da plataforma JAVA é essencial para enxergar os motivos que levaram ao sucesso.

Quais eram os maiores problemas quando programava na década de 90?

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

A linguagem Java resolve bem esses problemas, que até então apareciam com frequência nas outras linguagens.

Alguns desses problemas foram particularmente atacados porque uma das grandes motivações para a criação da plataforma Java era de que essa linguagem fosse usada em pequenos dispositivos, como tvs, videocassetes, aspiradores, liquidificadores e outros. Apesar disso a linguagem teve seu lançamento focado no uso em clientes web (browsers) para rodar pequenas aplicações (**applets**). Hoje em dia esse não é o grande mercado do Java: apesar de ter sido idealizado com um propósito e lançado com outro, o Java ganhou destaque no lado do servidor.

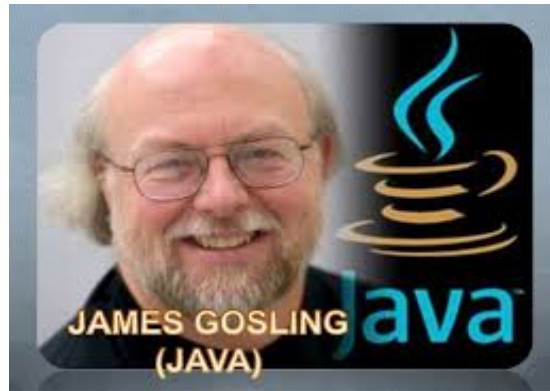
O Java foi criado pela antiga Sun Microsystems e mantida através de um comitê (<http://www.jcp.org>). Seu site principal era o java.sun.com, e java.com um site mais institucional, voltado ao consumidor de produtos e usuários leigos, não desenvolvedores. Com a compra da Sun pela Oracle em 2009, muitas URLs e nomes tem sido trocados para refletir a marca da Oracle. A página principal do Java é: <http://www.oracle.com/technetwork/java/>

No Brasil, diversos grupos de usuários se formaram para tentar disseminar o conhecimento da linguagem.

Um deles é o **GUJ** (<http://www.guj.com.br>), uma comunidade virtual com artigos, tutoriais e fórum para tirar dúvidas, o maior em língua portuguesa com mais de cem mil usuários e um milhão de mensagens.

1.1 - História do JAVA

O Java foi criado em 1991 por James Gosling da Sun Microsystems. Inicialmente chamada OAK (Carvalho), em homenagem à uma árvore de janelas do Gosling, seu nome foi mudado para Java devido a existência de uma linguagem com o nome OAK.



A motivação original do Java era a necessidade de uma linguagem independente de plataforma que podia ser utilizada em vários produtos eletrônicos, tais como torradeiras e Refrigeradores. Um dos primeiros projetos desenvolvidos utilizando Java era um controle remoto pessoal chamado *7 (Star Seven).



A Sun criou um time (conhecido como Green Team) para desenvolver inovações tecnológicas em 1992. Esse time foi liderado por James Gosling, considerado o pai do Java. O time voltou com a ideia de criar um interpretador (já era uma máquina virtual, veremos o que é isso mais a frente) para pequenos dispositivos, facilitando a reescrita de software para aparelhos eletrônicos, como vídeo cassete, televisão e aparelhos de TV a cabo.

A ideia não deu certo. Tentaram fechar diversos contratos com grandes fabricantes de eletrônicos, como Panasonic, mas não houve êxito devido ao conflito de interesses e custos. Hoje, sabemos que o Java domina o mercado de aplicações para celulares com mais de 2.5 bilhões de dispositivos compatíveis, porém em 1994 ainda era muito cedo para isso.

Com o advento da web, a Sun percebeu que poderia utilizar a ideia criada em 1992 para rodar pequenas aplicações dentro do browser. A semelhança era que na internet havia uma grande quantidade de sistemas operacionais e browsers, e com isso seria grande vantagem poder programar numa única linguagem, independente da plataforma. Foi aí que o Java 1.0 foi lançado: focado em transformar o browser de apenas um cliente magro (thin client ou terminal burro) em uma aplicação que possa também realizar operações avançadas, e não apenas renderizar html.

Os applets deixaram de ser o foco da Sun, e nem a Oracle nunca teve interesse. É curioso notar que a tecnologia Java nasceu com um objetivo em mente, foi lançado com outro, mas, no final, decolou mesmo no desenvolvimento de aplicações do lado do servidor. Sorte? Há hoje o Java FX, tentando dar força para o Java não só no desktop, mas como aplicações ricas na web, mas muitos não acreditam que haja espaço para tal, considerando o destino de tecnologias como Adobe Flex e Microsoft Silverlight.

Você pode ler a história da linguagem Java em: <http://www.java.com/en/javahistory/>
E um vídeo interessante: <http://tinyurl.com/histjava>. Em 2009 a Oracle comprou a Sun, fortalecendo a marca. A Oracle sempre foi, junto com a IBM, uma das empresas que mais investiram e fizeram negócios através do uso da plataforma Java. Em 2011 surge a versão Java 7 com algumas pequenas mudanças na linguagem, 2016 versão Java 8 e logo teremos o Java 9.

1.2 - Máquina Virtual

A **Máquina Virtual Java** é uma máquina imaginária que é implementada através de um software emulador em uma máquina real. A JVM provê especificações de plataforma de hardware na qual compila-se todo código de tecnologia Java. Essas especificações permitem que o software Java seja uma plataforma independente, pois a compilação é feita por uma máquina genérica conhecida como JVM.

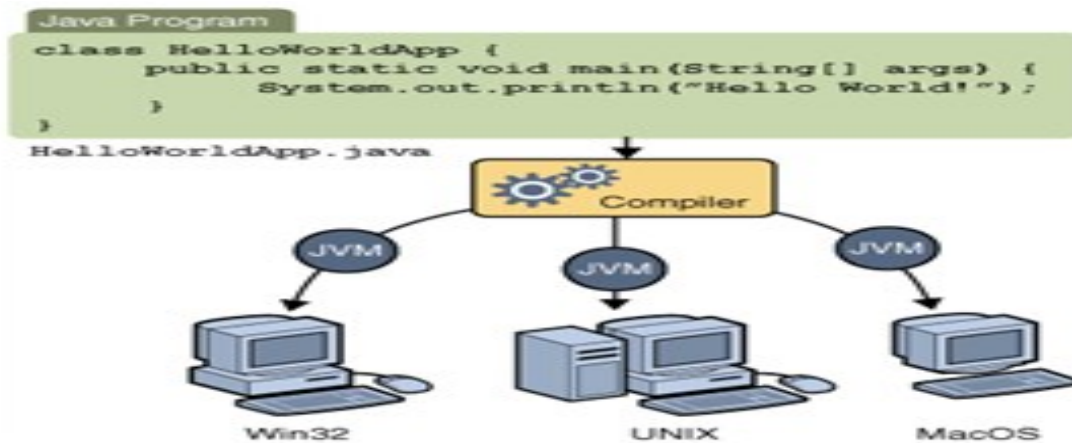
O **bytecode** é uma linguagem de máquina especial que pode ser entendida pela **Máquina Virtual Java (JVM)**. O bytecode é independente de qualquer hardware de computador particular. Assim, qualquer computador com o interpretador Java pode executar um programa Java compilado, não importando em que tipo de computador o programa foi compilado.

O código fonte é compilado para código de máquina específico de uma plataforma e sistema operacional.

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, e assim por diante, caso a gente queira que esse nosso software possa ser utilizado em várias plataformas. Esse é o caso de aplicativos como o OpenOffice, Firefox e outros.

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 “telas”. A biblioteca de interface gráfica do Windows é bem diferente das do Linux: como criar então uma aplicação que rode de forma parecida nos dois sistemas operacionais?

Precisamos reescrever um mesmo pedaço da aplicação para diferentes sistemas operacionais, já que eles não são compatíveis. Já o Java utiliza do conceito de **máquina virtual**, onde existe, entre o sistema operacional e a aplicação, uma camada extra responsável por “traduzir” - mas não apenas isso - o que sua aplicação deseja fazer para as respectivas chamadas do sistema operacional onde ela está rodando no momento:



Dessa forma, a maneira com 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.

Repare que uma máquina virtual é um conceito bem mais amplo que o de um interpretador. Como o próprio nome diz, uma máquina virtual é como um “computador de mentira”: tem tudo que um computador tem. Em outras palavras, ela é responsável por gerenciar memória, threads, a pilha de execução, etc.

Sua aplicação roda sem nenhum envolvimento como sistema operacional! Sempre conversando apenas com a **Java Virtual Machine** (JVM). Essa característica é interessante: como tudo passa pela JVM, ela pode tirar métricas, decidir onde é melhor alocar a memória, entre outros. Uma JVM isola totalmente a aplicação do sistema operacional. Se uma JVM termina abruptamente, só as aplicações que estavam rodando nela irão terminar: isso não afetará outras JVMs que estejam rodando no mesmo computador, nem afetará o sistema operacional.

Essa camada de isolamento também é interessante quando pensamos em um servidor que não pode se sujeitar a rodar código que possa interferir na boa execução de outras aplicações. Essa camada, a máquina virtual, não entende código java, ela entende um código de máquina específico. Esse código de máquina é gerado por um compilador java, como o **javac**, e é conhecido por “**bytecode**”, pois existem menos de 256 códigos de operação dessa linguagem, e cada “opcode” gasta um byte. 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 JVM.

2 – INSTALANDO O JAVA

Como vimos antes, a VM é apenas uma especificação e devemos baixar uma implementação. Há muitas empresas que implementam uma VM, como a própria Oracle, a IBM, a Apache e outros.

A da Oracle é a mais usada e possui versões para Windows, Linux e Solaris. Você pode baixar o SDK acessando: <http://www.oracle.com/technetwork/java/>

Nesta página da Oracle, você deve escolher o Java SE, dentro dos top downloads. Depois, escolha o JDK e seu sistema operacional. Para instalar o JDK no Windows, primeiro baixe-o no site da Oracle. É um simples arquivo executável que contém o Wizard de instalação: <http://www.oracle.com/technetwork/java/>



- 1) Dê um clique duplo no arquivo `jdk-<versão>-windows-i586-p.exe` e espere até ele entrar no wizard de instalação.



- 2) Aceite os próximos dois passos clicando em Next. Após um tempo, o instalador pedirá para escolher em que diretório instalar o SDK. Pode ser onde ele já oferece como padrão. Anote qual foi o diretório escolhido, vamos utilizar esse caminho mais adiante. A cópia de arquivos iniciará:

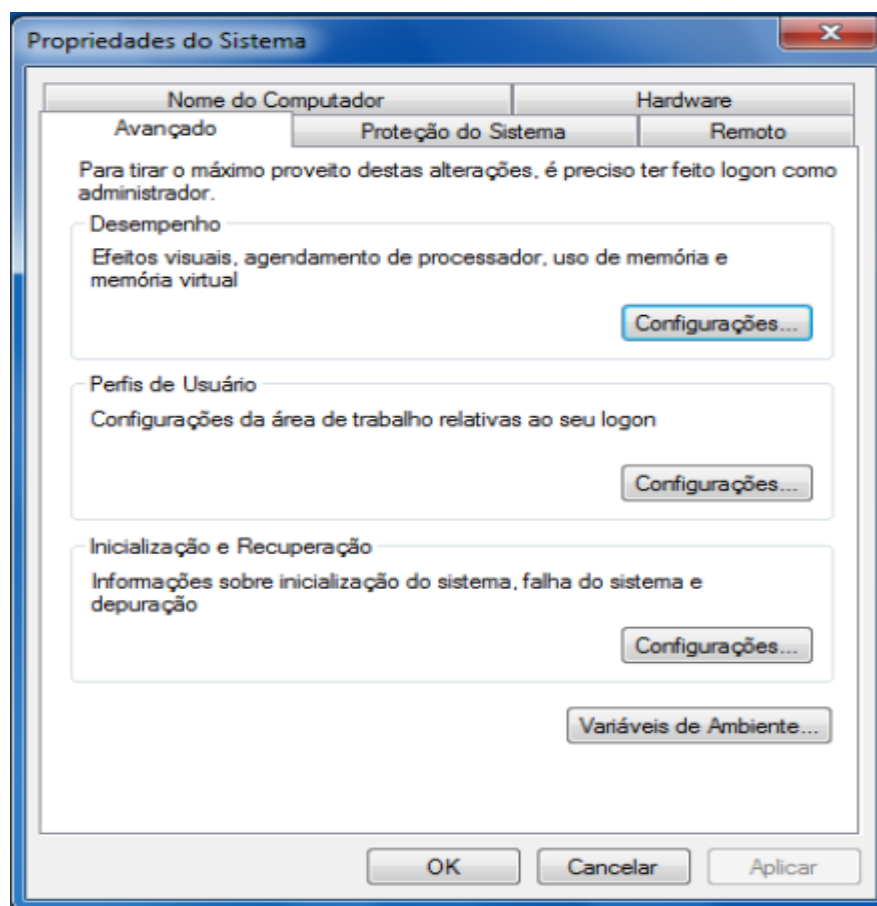


3) O instalador instalará também o JavaFX 6. Após isso, você será direcionado à uma página onde você pode, opcionalmente, criar uma conta na Oracle para registrar sua instalação.

2.1 – Configurando o ambiente

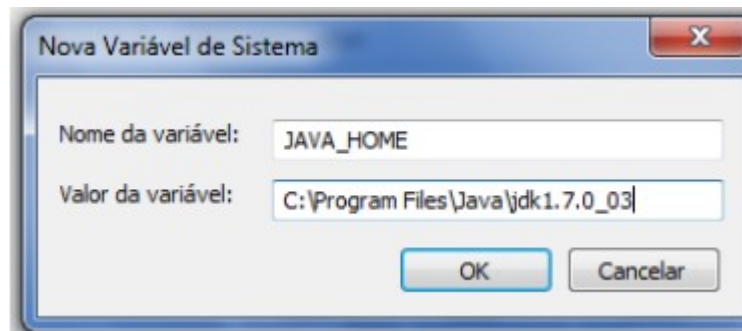
Precisamos configurar algumas variáveis de ambiente após a instalação, para que o compilador seja acessível via linha de comando. Caso você vá utilizar diretamente o Eclipse, provavelmente não será necessário realizar esses passos.

- 1) Clique com o botão direito em cima do ícone Computador e selecione a opção Propriedades.
- 2) Escolha a aba “Configurações Avançadas de Sistema” e depois clique no botão “Variáveis de Ambiente”.



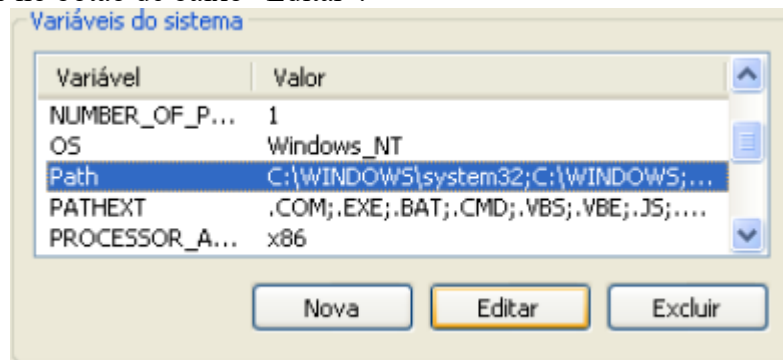
3) Nesta tela, você verá, na parte de cima, as variáveis de ambiente do usuário corrente e, embaixo, as variáveis de ambiente do computador (servem para todos os usuários). Clique no botão Novo... da parte de baixo.

4) Em Nome da Variável digite JAVA_HOME e, em valor da variável, digite o caminho que você utilizou na instalação do Java. Provavelmente será algo como: C:\Program Files\Java\jdk1.7.0_03:

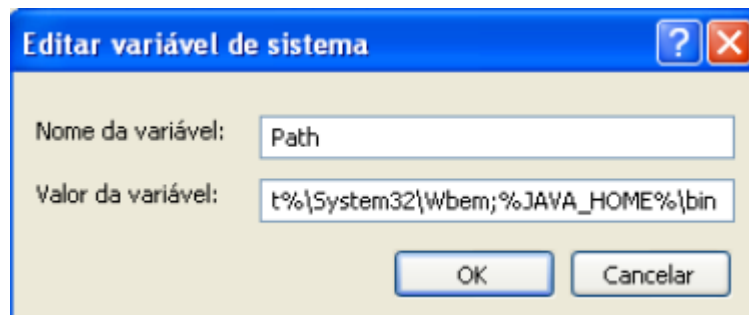


Clique em Ok.

5) Não vamos criar outra variável, mas sim alterar. Para isso, procure a variável PATH, ou Path (dá no mesmo), e clique no botão de baixo “Editar”.

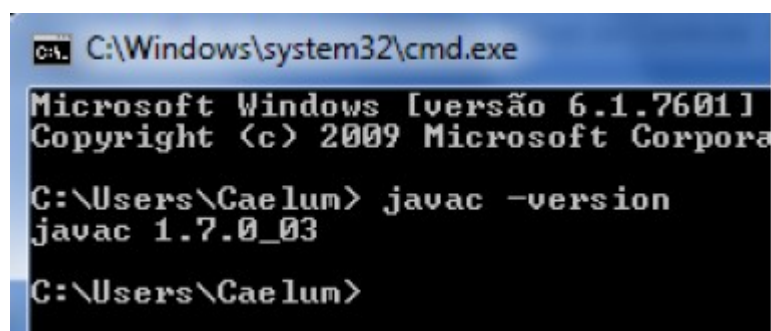


6) Não altere o nome da variável! Deixe como está e adicione no final do valor ;%JAVA_HOME%\bin, não esqueça do ponto-e-vírgula - assim, você está adicionando mais um caminho à sua variável Path.



7) Abra o prompt, indo em Iniciar, Executar e digite cmd.

8) No console, digite javac -version. O comando deve mostrar a versão do Java Compiler e algumas opções.



Você pode seguir para a instalação do Eclipse, conforme visto no seu capítulo, ou utilizar um editor de texto simples como o bloco de notas para os primeiros capítulos de apostila. Qualquer dúvida, não hesite de postá-la no Grupo de Usuários Java, em www.guj.com.br.

2 – COMPILANDO O PRIMEIRO PROGRAMA

Vamos para o nosso primeiro código! O programa que imprime uma linha simples. Para mostrar uma linha, podemos fazer:

```
System.out.println("Minha primeira aplicação Java!");
```

Mas esse código não será aceito pelo compilador java. O Java é uma linguagem bastante burocrática, e precisa de mais do que isso para iniciar uma execução. Veremos os detalhes e os porquês durante os próximos capítulos. O mínimo que precisaríamos escrever é algo como:

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

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 Oracle, chamado `javac`, gere o `bytecode` correspondente ao 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.

2.1 – 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.

2.2 – O que aconteceu

```
class MeuPrograma {
    public static void main(String[] args) {
        // miolo do programa começa aqui!
        System.out.println("Minha primeira aplicação Java!!");
        // fim do miolo do programa
    }
}
```

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

importam para nós nesse momento. Mas devemos saber que toda aplicação Java começa por um ponto de entrada, e este ponto de entrada é o método main.

Ainda não sabemos o que é método, mas veremos no capítulo. Até lá, não se preocupe com essas declarações.

Sempre que um exercício for feito, o código que nos importa 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.

2.3 – Para saber mais : Como é o Bytecode ?

O `MeuPrograma.class` gerado não é legível por seres humanos (não que seja impossível). Ele está escrito no formato que a virtual machine sabe entender e que foi especificado que ela entendesse. É como um assembly, escrito para esta máquina em específico. Podemos ler os mnemônicos utilizando a ferramenta `javap` que acompanha o JDK:

```
javap -c MeuPrograma
```

E a saída:

```
MeuPrograma();
```

Code:

```
0: aload_0
1: invokespecial #1; //Method java/lang/Object."<init>":()V
4: return
```

```
public static void main(java.lang.String[]);
```

Code:

```
0: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc #3; //String Minha primeira aplicação Java!!
5: invokevirtual #4; //Method java/io/PrintStream.println:
    (Ljava/lang/String;)V
8: return
}
```

É o código acima, que a JVM sabe ler. É o “código de máquina”, da máquina virtual. Um bytecode pode ser revertido para o .java original (com perda de comentários e nomes de variáveis locais). Caso seu software vá virar um produto de prateleira, é fundamental usar um ofuscador no seu código, que vai embaralhar classes, métodos e um monte de outros recursos (indicamos o <http://proguard.sf.net>).

2.4 – Exercícios : Modificando o Hello World

- 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`.

2.5 – O quê pode dar errado ?

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

Código:

```
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. Repare que o compilador é explícito em dizer que a linha 4 é a com problemas. Outros erros de compilação podem ocorrer se você escreveu palavras chaves (as 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:

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

- Se não colocar o método main como public:

Main method not public.

Por exemplo:

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

2.6 – 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?

3 - Variáveis Primitivas e Controles de Fluxo

3.1 – Declarando e usando 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:

```
tipoDaVariavel nomeDaVariavel;
```

Por exemplo, é possível ter uma idade que guarda um número inteiro:

```
int idade;
```

Com isso, você declara a variável idade, que passa a existir a partir daquela linha. Ela é do tipo int, que guarda um número inteiro. A partir daí, você pode usá-la, primeiramente atribuindo valores.

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

```
idade = 15;
```

3.2 – 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**.

```
// calcula a idade no ano seguinte
int idadeNoAnoQueVem;
idadeNoAnoQueVem = idade + 1;
```

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

```
int idade = 15;
```

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 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
```

3.3 – Como rodar esses códigos ?

Você deve colocar esses trechos de código dentro do bloco `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:

```
class TestaIdade {
    public static void main(String[] args) {
        // imprime a idade
        int idade = 20;
        System.out.println(idade);
        // gera uma idade no ano seguinte
        int idadeNoAnoQueVem;
        idadeNoAnoQueVem = idade + 1;
        // imprime a idade
        System.out.println(idadeNoAnoQueVem);
    }
}
```

```

    }
}

```

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

```

double pi = 3.14;
double x = 5 * 10;

```

O tipo `boolean` armazena um valor verdadeiro ou falso, e só: nada de números, palavras ou endereços, como em algumas outras linguagens.

```

boolean verdade = true;

```

`true` e `false` são palavras reservadas do Java. É comum que um `boolean` seja determinado através de uma **expressão booleana**, isto é, um trecho de código que retorna um booleano, como o exemplo:

```

int idade = 30;
boolean menorDeIdade = idade < 18;

```

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);

```

Variáveis do tipo `char` são pouco usadas no dia a dia. Veremos mais a frente o uso das `Strings`, que usamos constantemente, porém estas não são definidas por um tipo primitivo.

3.4 – Tipos primitivos e valores

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 muda de valor? Não, pois o valor de um tipo primitivo sempre é copiado. Apesar da linha ó fazer `j = i`, a partir desse momento essas variáveis não tem relação nenhuma: o que acontece com uma, não reflete em nada com a outra.

3.5 – 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.

3.6 – Exercícios : Variáveis e Tipos Primitivos

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, foram gastos 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:

- a) Crie uma classe chamada BalancoTrimestral com um bloco main, como nos exemplos anteriores;
- b) Dentro do main (o miolo do programa), declare uma variável inteira chamada gastosJaneiro e inicialize-a com 15000;
- c) Crie também as variáveis gastosFevereiro e gastosMarco, inicializando-as com 23000 e 17000, respectivamente, utilize uma linha para cada declaração;
- d) Crie uma variável chamada gastosTrimestre e inicialize-a com a soma das outras 3 variáveis:

```
int gastosTrimestre = gastosJaneiro + gastosFevereiro + gastosMarco;
```
- e) Imprima a variável gastosTrimestre.

2) Adicione código (sem alterar as linhas que já existem) na classe anterior para imprimir a média mensal de gasto, criando uma variável mediaMensal junto com uma mensagem. Para isso, concatene a String com o valor, usando "Valor da média mensal = "+ mediaMensal.

3.7 – 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 porque 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.

Às vezes, precisamos que um número quebrado seja arredondado e armazenado em 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 de i agora é 3. O mesmo ocorre entre valores int e long.

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

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

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

3.8 – Casos não 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, que pode ser maiúscula ou minúscula, indica que aquele literal deve ser tratado como float.

Outro caso, que é mais comum:

```
double d = 5;
float f = 3;
float x = f + (float) d;
```

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, uma observação: no mínimo, o Java armazena o resultado em um int, na hora de fazer as contas. 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.

3.9 – Casting possíveis

Abaixo estão relacionados todos os casts possíveis na linguagem Java, mostrando a conversão 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:	byte	short	char	int	long	float	double
DE:							
byte	----	<i>Impl.</i>	(char)	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
short	(byte)	----	(char)	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
char	(byte)	(short)	----	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
int	(byte)	(short)	(char)	----	<i>Impl.</i>	<i>Impl.</i>	<i>Impl.</i>
long	(byte)	(short)	(char)	(int)	----	<i>Impl.</i>	<i>Impl.</i>
float	(byte)	(short)	(char)	(int)	(long)	----	<i>Impl.</i>
double	(byte)	(short)	(char)	(int)	(long)	(float)	----

3.10 – 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

3.11 – O IF e o ELSE

A sintaxe do if no Java é a seguinte:

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

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");
}
```

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

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

Você pode concatenar expressões booleanas através dos operadores lógicos "E" e "OU". O "E" é representado pelo && e o "OU" é representado pelo ||.

Um exemplo seria verificar se ele tem menos de 18 anos e se ele não é amigo do dono:

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 && amigoDoDono == false) {
    System.out.println("Não pode entrar");
}
else {
    System.out.println("Pode entrar");
}
```

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

```
int idade = 15;
boolean amigoDoDono = true;
if (idade < 18 && !amigoDoDono) {
    System.out.println("Não pode entrar");
}
else {
    System.out.println("Pode entrar");
}
```

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 = dentro de um if 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");
}
```

```
}
```

3.12 – O While

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

```
int idade = 15;
while (idade < 18) {
    System.out.println(idade);
    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 não o fará imprimir 18.

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

Já o while acima imprime de 0 a 9.

3.13 - O For

Outro comando de **loop** extremamente utilizado é o for. A ideia 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 fiquem mais legíveis, 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) {
    System.out.println("olá!");
    i = i + 1;
}
```

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.

3.14 – Pós incremento ++

`i = i + 1` pode realmente ser substituído por `i++` quando isolado, porém, em alguns casos, temos essa instrução envolvida em, por exemplo, uma atribuição:

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

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

O operador `++`, quando vem após a 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:

```
int i = 5;
int x = ++i; // aqui x valera 6
```

3.15 – 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.

```
for (int i = x; i < y; i++) {
    if (i % 19 == 0) {
        System.out.println("Achei um número divisível por 19 entre x e y");
        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`. Da mesma maneira, é possível obrigar o loop a executar o próximo laço. Para isso usamos a palavra chave `continue`.

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

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

3.16 – 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
```

O **escopo da variável** é o nome dado ao trecho de código em que aquela variável existe e onde é possível acessá-la. Quando abrimos um novo bloco com as chaves, as variáveis declaradas ali dentro **só valem até o fim daquele bloco**.

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

// a partir daqui ela existe
while (condicao) {
    // o i ainda vale aqui
    int j = 7;
    // o j passa a existir
}
// aqui o j não existe mais, mas o i continua dentro do escopo
```

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. O mesmo vale para um *if*:

```
if (algumBooleano) {
    int i = 5;
}
else {
    int i = 10;
}
System.out.println(i); // cuidado!
```

Aqui a variável *i* não existe fora do *if* e do *else*! Se você declarar a variável antes do *if*, vai haver outro erro de compilação: dentro do *if* e do *else* a variável está sendo redeclarada! Então o código para compilar e fazer sentido fica:

```
int i;
if (algumBooleano) {
    i = 5;
}
else {
    i = 10;
}
System.out.println(i);
```

Uma situação parecida pode ocorrer com o *for*:

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

Neste for, a variável `i` morre ao seu término, não podendo ser acessada de fora do for, gerando um erro de compilação. Se você realmente quer acessar o contador depois do loop terminar, precisa de algo como:

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

3.17 – 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
    }
}
```

3.18 – Para saber 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 ifs 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) Além dos operadores de incremento, existem os de decremento, como `--i` e `i--`. Além desses, você pode usar instruções do tipo `i += x` e `i -= x`, o que essas instruções fazem? Teste.

3.19 – Exercícios : Fixação de sintaxe

Mais exercícios de fixação de sintaxe. Para quem já conhece um pouco de Java, pode ser muito simples; mas recomendamos fortemente que você faça os exercícios para se acostumar com erros de compilação, mensagens do `javac`, convenção de código, etc...

Apesar de extremamente simples, precisamos praticar a sintaxe que estamos aprendendo. Para cada exercício, crie um novo arquivo com extensão **.java**, e declare aquele estranho cabeçalho, dando nome a uma classe e com um bloco main dentro dele:

```
class ExercicioX {
    public static void main(String[] args) {
        // seu exercício vai aqui
    }
}
```

Não copie e cole de um exercício já existente! Aproveite para praticar.

- 1) Imprima todos os números de 150 a 300.
- 2) Imprima a soma de 1 até 1000.
- 3) Imprima todos os múltiplos de 3, entre 1 e 100.

4 – Orientação a objetos básica

4.1 – Motivação : programas do paradigma procedural

Orientação a 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 vai validá-lo, como no pseudocódigo abaixo:

```
cpf = formulario->campo_cpf
valida(cpf)
```

Alguém te obriga a sempre validar esse CPF? Você pode, inúmeras vezes, esquecer de chamar esse validador. Mais: considere que você tem 50 formulários e precise validar em todos eles o CPF. Se sua equipe tem 3 programadores trabalhando nesses formulários, quem fica responsável por essa validação? Todos!

A situação pode piorar: na entrada de um novo desenvolvedor, precisaríamos avisá-lo que sempre devemos validar o cpf de um formulário. É nesse momento que nascem aqueles guias de programação para o desenvolvedor que for entrar nesse projeto - às vezes, é um documento enorme. Em outras palavras, **todo** desenvolvedor precisa ficar sabendo de uma quantidade enorme de informações, que, na maioria das vezes, não está realmente relacionado à sua parte no sistema, mas ele **precisa** ler tudo isso, resultando um entrave muito grande!

Outra situação onde ficam claros os problemas da programação procedural, é quando nos encontramos na necessidade de ler o código que foi escrito por outro desenvolvedor e descobrir como

ele funciona internamente. Um sistema bem encapsulado não deveria gerar essa necessidade. Em um sistema grande, simplesmente não temos tempo de ler todo o código existente.

Considerando que você não erre nesse ponto e que sua equipe tenha uma comunicação muito boa (perceba que comunicação excessiva pode ser prejudicial e atrapalhar o andamento), ainda temos outro problema: imagine que, em todo formulário, você também quer que a idade do cliente seja validada - o cliente precisa ter mais de 18 anos. Vamos ter de colocar um if... mas onde? Espalhado por todo seu código... Mesmo que se crie outra função para validar, precisaremos incluir isso nos nossos 50 formulários já existentes. Qual é a chance de esquecermos em um deles? É muito grande.

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

Melhor ainda seria se conseguíssemos mudar essa validação e os outros programadores nem precisassem ficar sabendo disso. Em outras palavras, eles criariam formulários e um único programador seria responsável pela validação: os outros nem sabem da existência desse trecho de código. Impossível? Não, o paradigma da orientação a objetos facilita tudo isso.

O problema do paradigma procedural é que não existe uma forma simples de criar conexão forte entre dados e funcionalidades. No paradigma orientado a objetos é muito fácil ter essa conexão através dos recursos da própria linguagem.

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, **encapsulando** a lógica de negócios.

Outra enorme vantagem, onde você realmente vai economizar montanhas de código, é o **polimorfismo das referências**, que veremos em um posterior capítulo. Nos próximos capítulos, conseguiremos enxergar toda essa vantagem, mas, primeiramente é necessário conhecer um pouco mais da sintaxe e da criação de tipos e referências em Java.

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 ideia aqui é generalizarmos alguma informação, juntamente com funcionalidades que toda conta deve ter.

O que toda conta tem e é importante para nós?

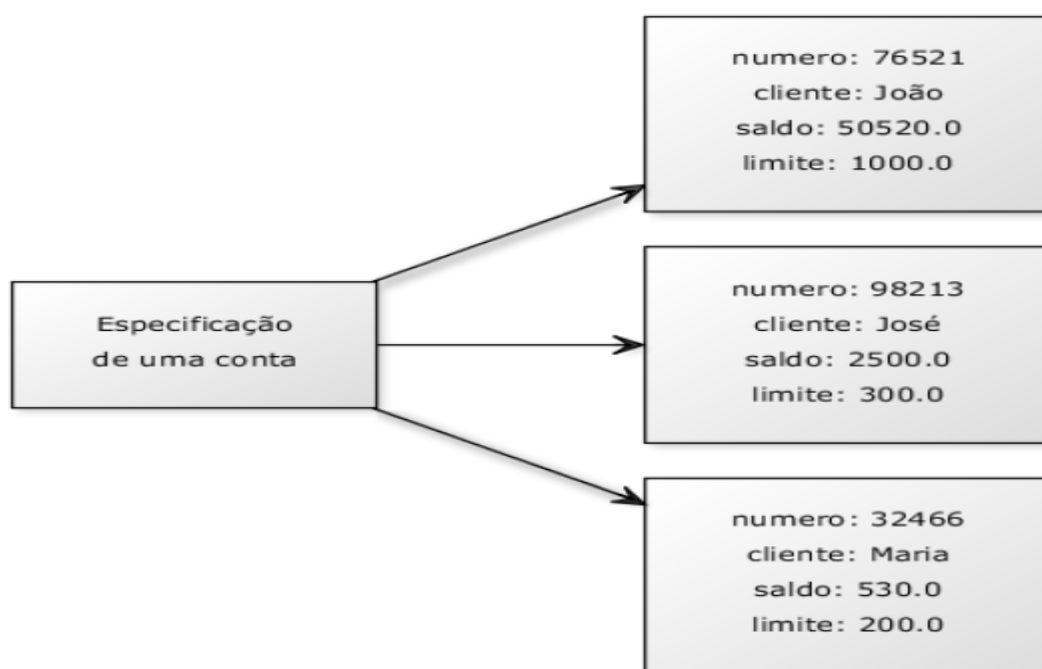
- número da conta
- nome do dono da conta
- saldo
- limite

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 conta, para poder acessar o que ela tem, e pedir a ela que faça algo.



Repare na figura: apesar do papel do lado esquerdo especificar uma Conta, essa especificação é uma Conta? Nós depositamos e sacamos dinheiro desse papel? Não. Utilizamos a especificação da Conta para poder criar instâncias que realmente são contas, onde podemos realizar as operações que criamos.

Apesar de declararmos que toda conta tem um saldo, um número e uma agência no pedaço de papel (como à esquerda na figura), são nas instâncias desse projeto que realmente há espaço para armazenar esses valores. Ao projeto da conta, isto é, a definição da conta, damos o nome de **classe**.

Ao que podemos construir a partir desse projeto, as contas de verdade, damos o nome de **objetos**.

A palavra **classe** vem da taxonomia da biologia. Todos os seres vivos de uma mesma **classe** biológica têm uma série de **atributos** e **comportamentos** em comum, mas não são iguais, podem variar nos valores desses **atributos** e como realizam esses **comportamentos**.

Homo Sapiens define um grupo de seres que possuem características em comum, porém a definição (a ideia, o conceito) de um **Homo Sapiens** é um ser humano? Não. Tudo está especificado na **classe** Homo Sapiens, mas se quisermos mandar alguém correr, comer, pular, precisaremos de uma instância de **Homo Sapiens**, ou então de um **objeto** do tipo **Homo Sapiens**.

Um outro exemplo: uma receita de bolo. A pergunta é certa: você come uma receita de bolo? Não. Precisamos **instanciá-la**, criar um **objeto** bolo a partir dessa especificação (a classe) para utilizá-la. Podemos criar centenas de bolos a partir dessa classe (a receita, no caso), eles podem ser bem semelhantes, alguns até idênticos, mas são **objetos** diferentes.

Podemos fazer milhares de analogias semelhantes. A planta de uma casa é uma casa? Definitivamente não. Não podemos morar dentro da planta de uma casa, nem podemos abrir sua porta ou pintar suas paredes. Precisamos, antes, construir instâncias a partir dessa planta. Essas instâncias, sim, podemos pintar, decorar ou morar dentro.

Pode parecer óbvio, mas a dificuldade inicial do paradigma da orientação a objetos é justo saber distinguir o que é classe e o que é objeto. É comum o iniciante utilizar, obviamente de forma errada, essas duas palavras como sinônimos.

4.3 – Uma classe em Java

Vamos começar apenas com o que uma Conta tem, e não com o que ela faz (veremos logo em seguida). Um tipo desses, como o especificado de Conta acima, pode ser facilmente traduzido para Java:

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

String é uma classe em Java. Ela guarda uma cadeia de caracteres, uma frase completa. Como estamos ainda aprendendo o que é uma classe, entenderemos com detalhes a classe String apenas em capítulos posteriores.

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 fazíamos quando tinha aquele main. Quando uma variável é declarada diretamente dentro do escopo da classe, é chamada de variável de objeto, ou atributo.

4.4 – Criando e usando um objeto

Já temos uma classe em Java que especifica o que todo objeto dessa classe deve ter. Mas como usá-la? Além dessa classe, ainda teremos o **Programa.java** e a partir dele é que vamos utilizar a classe Conta.

Para criar (construir, instanciar) uma Conta, basta usar a palavra chave new. Devemos utilizar também os parênteses, que descobriremos o que fazem exatamente em um capítulo posterior:

```
class Programa {
    public static void main(String[] args) {
        new Conta();
    }
}
```

Bem, o código acima cria um objeto do tipo Conta, mas como acessar esse objeto que foi criado? Precisamos ter alguma forma de nos referenciarmos a esse objeto. Precisamos de uma variável:

```
class Programa {
    public static void main(String[] args) {
        Conta minhaConta;
        minhaConta = new Conta();
    }
}
```

Pode parecer estranho escrevermos duas vezes Conta: uma vez na declaração da variável e outra vez no uso do new. Mas há um motivo, que em breve entenderemos.

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

```
class Programa {
    public static void main(String[] args) {
        Conta minhaConta;
        minhaConta = new Conta();
        minhaConta.dono = "Duke";
        minhaConta.saldo = 1000.0;
        System.out.println("Saldo atual: " + minhaConta.saldo);
    }
}
```

É importante fixar que o ponto foi utilizado para acessar algo em minhaConta. A minhaConta pertence ao Duke, e tem saldo de mil reais.

4.5 – Métodos

Dentro da classe, também declararemos o que cada conta faz e como isto é feito - os comportamentos que cada classe tem, isto é, o que ela faz. Por exemplo, de que maneira que uma Conta saca dinheiro? Especificaremos isso dentro da própria classe Conta, e não em um local desatrelado das informações da própria Conta.

É por isso que essas “funções” são chamadas de **métodos**. Pois é a maneira de fazer uma operação com um objeto.

Queremos criar um método que **saca** uma determinada **quantidade** e não devolve **nenhuma informação** para quem acionar esse método:

```
class Conta {
    double salario;
    // ... outros atributos ...
    void saca(double quantidade) {
        double novoSaldo = this.saldo - quantidade;
        this.saldo = novoSaldo;
    }
}
```

A palavra chave `void` diz que, quando você pedir para a conta sacar uma quantia, nenhuma informação será enviada de volta a quem pediu.

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 ou local, pois, ao final da execução desse método, ela deixa de existir.

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. (veremos depois que é opcional).

Repare que, nesse caso, a conta pode estourar o limite fixado pelo banco. Mais para frente, evitaremos essa situação, e de uma maneira muito elegante. Da mesma forma, temos o método para depositar alguma quantia:

```
class Conta {
    // ... outros atributos e métodos ...
    void deposita(double quantidade) {
        this.saldo += quantidade;
    }
}
```

Observe que não usamos uma variável auxiliar e, além disso, usamos a abreviação `+=` para deixar o método bem simples. O `+=` soma quantidade ao valor antigo do saldo e guarda no próprio saldo, o valor resultante. Para mandar uma mensagem ao objeto e pedir que ele execute um método, também usamos o ponto. O termo usado para isso é **invocação de método**.

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

```
class TestaAlgunsMetodos {
    public static void main(String[] args) {
        // criando a conta
        Conta minhaConta;
        minhaConta = new Conta();
        // alterando os valores de minhaConta
        minhaConta.dono = "Duke";
        minhaConta.saldo = 1000;

        // saca 200 reais
        minhaConta.saca(200);

        // deposita 500 reais
        minhaConta.deposita(500);
        System.out.println(minhaConta.saldo);
    }
}
```

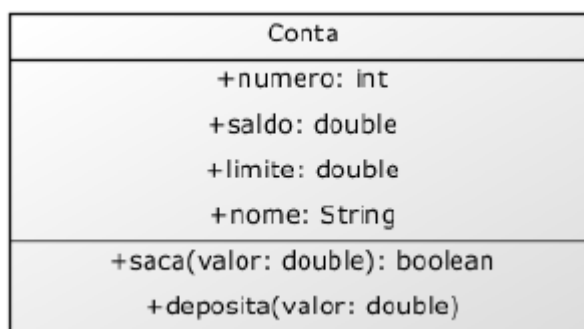
Uma vez que seu saldo inicial é 1000 reais, se sacarmos 200 reais, depositarmos 500 reais e imprimirmos o valor do saldo, o que será impresso?

4.6 – Métodos com retorno

Um método sempre tem que definir o que retorna, nem que defina que não há retorno, como nos exemplos anteriores onde estávamos usando o void. Um método pode retornar um valor para o código que o chamou. No caso do nosso método `saca`, podemos devolver um valor booleano indicando se a operação foi bem sucedida.

```
class Conta {
    // ... outros métodos e atributos ...
    boolean saca(double valor) {
        if (this.saldo < valor) {
            return false;
        }
        else {
            this.saldo = this.saldo - valor;
            return true;
        }
    }
}
```

A declaração do método mudou! O método `saca` não tem void na frente. Isto quer dizer que, quando é 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);
if (consegui) {
    System.out.println("Consegui sacar");
} else {
    System.out.println("Não consegui sacar");
}
```

Ou então, posso eliminar a variável temporária, se desejado:

```
minhaConta.saldo = 1000;
if (minhaConta.saca(2000)) {
    System.out.println("Consegui sacar");
} else {
    System.out.println("Não consegui sacar");
}
```

```
}
```

Mais adiante, veremos que algumas vezes é mais interessante lançar uma exceção (exception) nesses casos. Meu programa pode manter na memória não apenas uma conta, como mais de uma:

```
class TestaDuasContas {
    public static void main(String[] args) {
        Conta minhaConta;
        minhaConta = new Conta();
        minhaConta.saldo = 1000;

        Conta meuSonho;
        meuSonho = new Conta();
        meuSonho.saldo = 1500000;
    }
}
```

4.7 – Objetos são acessados por referências

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**.

É por esse motivo que, diferente dos tipos primitivos como int e long, precisamos dar new depois de declarada a variável:

```
public static void main(String args[]) {
    Conta c1;
    c1 = new Conta();

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

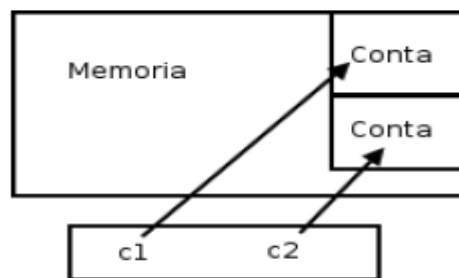
O correto aqui, é dizer que c1 se refere a um objeto. **Não é correto** dizer que c1 é um objeto, pois c1 é uma variável referência, apesar de, depois de um tempo, os programadores Java falarem “Tenho um **objeto c** do tipo **Conta**”, mas apenas para encurtar a frase “Tenho uma **referência c** a um **objeto** do tipo **Conta**”.

Basta lembrar que, em Java, **uma variável nunca é um objeto**. Não há, no Java, uma maneira de criarmos o que é conhecido como “objeto pilha” ou “objeto local”, pois todo objeto em Java, sem exceção, é acessado por uma variável referência.

Esse código nos deixa na seguinte situação:

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

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



Internamente, `c1` e `c2` vão guardar um número que identifica em que posição da memória aquela `Conta` se encontra. Dessa maneira, ao utilizarmos o “.” para navegar, o Java vai acessar a `Conta` que se encontra naquela posição de memória, e não uma outra. Para quem conhece, é parecido com um ponteiro, porém você não pode manipulá-lo como um número e nem utilizá-lo para aritmética, ela é tipada.

Um outro exemplo:

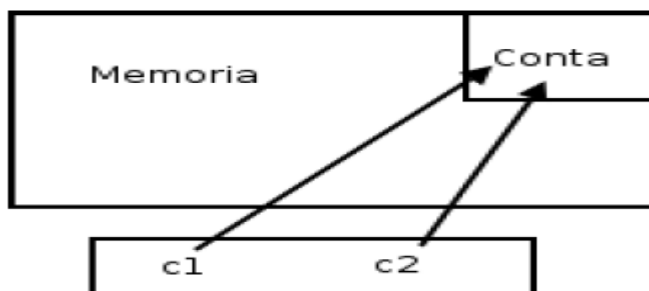
```
class TestaReferencias {  
    public static void main(String args[]) {  
        Conta c1 = new Conta();  
        c1.deposita(100);  
  
        Conta c2 = c1; // linha importante!  
        c2.deposita(200);  
  
        System.out.println(c1.saldo);  
        System.out.println(c2.saldo);  
    }  
}
```

Qual é o resultado do código acima? O que aparece ao rodar?

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**) de onde o objeto se encontra na memória principal.

Na memória, o que acontece nesse caso:

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

Quando fazemos `c2 = c1`, `c2` passa a fazer referência para o mesmo objeto que `c1` referencia nesse instante. Então, nesse código em específico, quando utilizamos `c1` ou `c2` estamos nos referindo exatamente ao **mesmo** objeto! Elas são duas referências distintas, porém apontam para o **mesmo** objeto! Compará-las com “`==`” vai nos retornar `true`, pois o valor que elas carregam é o mesmo!

Outra forma de perceber, é que demos apenas um `new`, então só pode haver um objeto `Conta` na memória. **Atenção:** não estamos discutindo aqui a utilidade de fazer uma referência apontar pro mesmo objeto que outra. Essa utilidade ficará mais clara quando passarmos variáveis do tipo referência como argumento para métodos.

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 a uma variável, essa variável passa a se referir para esse mesmo objeto.

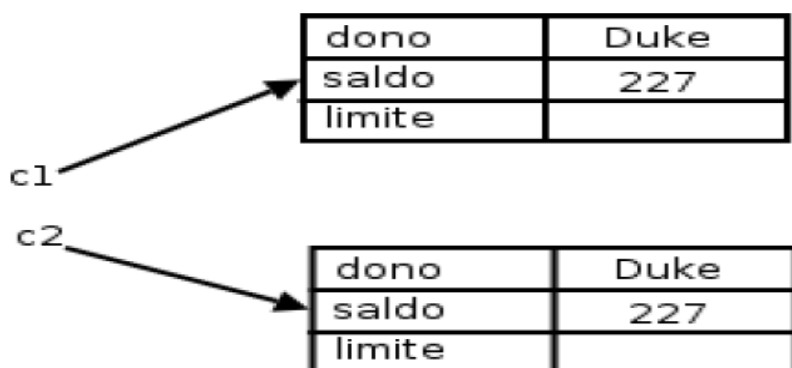
Podemos então ver outra situação:

```
public static void main(String args[]) {
    Conta c1 = new Conta();
    c1.dono = "Duke";
    c1.saldo = 227;

    Conta c2 = new Conta();
    c2.dono = "Duke";
    c2.saldo = 227;

    if (c1 == c2) {
        System.out.println("Contas iguais");
    }
}
```

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, elas 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 elas não são o mesmo objeto. 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 têm o mesmo conteúdo, você precisa comparar atributo por atributo. Veremos uma solução mais elegante para isso também.

4.8 – O método transfere()

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

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

```

class Conta {
    // atributos e métodos...
    void transfere(Conta destino, double valor) {
        this.saldo = this.saldo - valor;
        destino.saldo = destino.saldo + valor;
    }
}
  
```

Conta
+numero: int +saldo: double +limite: double +nome: String
+saca(valor: double): boolean +deposita(valor: double) +transfere(destino: Conta, valor: double)

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 métodos...
    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;
        }
    }
}

```

Conta
+numero: int
+saldo: double
+limite: double
+nome: String
+saca(valor: double): boolean
+deposita(valor: double)
+transfere(destino: Conta, valor: double): boolean

Quando passamos uma Conta como argumento, o que será que acontece na memória? Será que o objeto é clonado?

No Java, a passagem de parâmetro funciona como uma simples atribuição como no uso do “=”. Então, esse parâmetro vai copiar o valor da variável do tipo Conta que for passado como argumento. E qual é o valor de uma variável dessas? Seu valor é um endereço, uma referência, nunca um objeto. Por isso não há cópia de objetos aqui.

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

Perceba que o nome deste método poderia ser `transferePara` ao invés de só `transfere`. A chamada do método fica muito mais natural, é possível ler a frase em português que ela tem um sentido:

```
conta1.transferePara(conta2, 50);
```

A leitura deste código seria “Conta1 transfere para conta2 50 reais”.

4.9 – 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, valem false. Você também pode dar **valores default**, como segue:

```

class Conta {
    int numero = 1234;
    String dono = "Duke";
    String cpf = "123.456.789-10";
    double saldo = 1000;
    double limite = 1000;
}

```

Nesse caso, quando você criar uma conta, seus atributos já estão “populados” com esses valores colocados. Imagine que começemos a aumentar nossa classe Conta e adicionar nome, sobrenome e cpf do cliente dono da conta. Começaríamos a ter muitos atributos... e, se você pensar direito, uma Conta não tem nome, nem sobrenome nem cpf, quem tem esses atributos é um Cliente.

Então podemos criar uma nova classe e fazer uma composição. Seus atributos também podem ser referências para outras classes. Suponha a seguinte classe Cliente:

```
class Cliente {
    String nome;
    String sobrenome;
    String cpf;
}

class Conta {
    int numero;
    double saldo;
    double limite;
    Cliente titular;
}
```

E dentro do main da classe de teste:

```
class Teste {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        Cliente c = new Cliente();
        minhaConta.titular = c;
    }
}
```

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

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

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

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

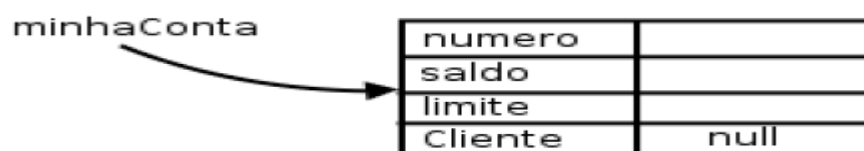
```
minhaConta.titular.nome = "Duke";
```

Um sistema orientado a objetos é um grande conjunto de classes que vai se comunicar, delegando responsabilidades para quem for mais apto a realizar determinada tarefa. A classe Banco usa a classe Conta que usa a classe Cliente, que usa a classe Endereco. Dizemos que esses objetos colaboram, trocando mensagens entre si. Por isso acabamos tendo muitas classes em nosso sistema, e elas costumam ter um tamanho relativamente curto.

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

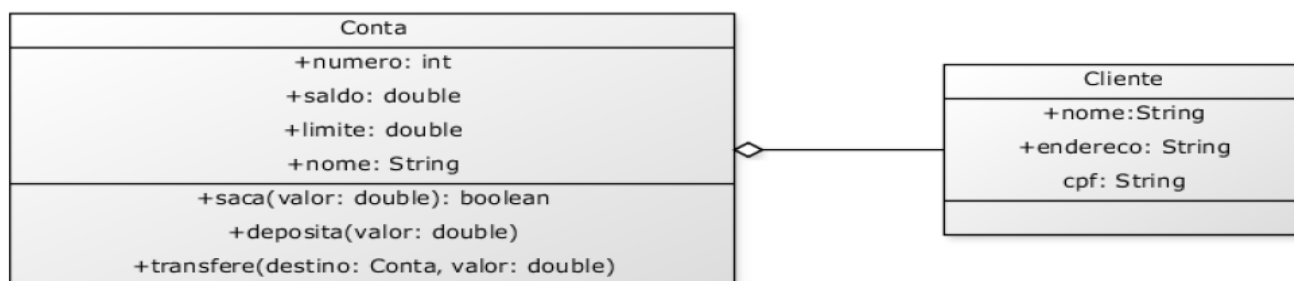
```
class Teste {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.titular.nome = "Manoel";
        // ...
    }
}
```

Quando damos new em um objeto, ele o inicializa 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 à frente). Da para perceber, então, que o new não traz um efeito cascata, a menos que você dê um valor default (ou use construtores, que também veremos mais a frente):

```
class Conta {
    int numero;
    double saldo;
    double limite;
    Cliente titular = new Cliente(); // quando chamarem new Conta,
    //havera um new Cliente para ele.
}
```



Com esse código, toda nova Conta criada já terá um novo Cliente associado, sem necessidade de instanciá-lo logo em seguida da instanciização 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.

Atenção: para quem não está acostumado com referências, pode ser bastante confuso pensar sempre em como os objetos estão na memória para poder tirar as conclusões de o que ocorrerá ao executar determinado código, por mais simples que ele seja. Com tempo, você adquire a habilidade de

rapidamente saber o efeito de atrelar as referências, sem ter de gastar muito tempo para isso. É importante, nesse começo, você estar sempre pensando no estado da memória. E realmente lembrar que, no Java "uma variável nunca carrega um objeto, e sim uma referência para ele" facilita muito.

4.10 – Para saber mais: Uma fábrica de carros.

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

```
class Carro {
    String cor;
    String modelo;
    double velocidadeAtual;
    double velocidadeMaxima;

    //liga o carro
    void liga() {
        System.out.println("O carro está ligado");
    }

    //acelera uma certa quantidade
    void acelera(double quantidade) {
        double velocidadeNova = this.velocidadeAtual + quantidade;
        this.velocidadeAtual = velocidadeNova;
    }

    //devolve a marcha do carro
    int pegaMarcha() {
        if (this.velocidadeAtual < 0) {
            return -1;
        }
        if (this.velocidadeAtual >= 0 && this.velocidadeAtual < 40) {
            return 1;
        }
        if (this.velocidadeAtual >= 40 && this.velocidadeAtual < 80) {
            return 2;
        }
        return 3;
    }
}
```

Vamos testar nosso Carro em um novo programa:

```
class TestaCarro {
    public static void main(String[] args) {
        Carro meuCarro;
        meuCarro = new Carro();
        meuCarro.cor = "Verde";
        meuCarro.modelo = "Fusca";
        meuCarro.velocidadeAtual = 0;
        meuCarro.velocidadeMaxima = 80;

        // liga o carro
        meuCarro.liga();
        // acelera o carro
        meuCarro.acelera(20);
        System.out.println(meuCarro.velocidadeAtual);
    }
}
```

Nosso carro pode conter também um Motor:

```
class Motor {
    int potencia;
    String tipo;
}

class Carro {
    String cor;
    String modelo;
    double velocidadeAtual;
    double velocidadeMaxima;
    Motor motor;
}
```

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

4.11 – Um pouco mais

1) Quando declaramos uma classe, um método ou um atributo, podemos dar o nome que quisermos, 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 às variáveis com letras minúsculas. É que existem **convenções de código**, dadas pela Oracle, para facilitar a legibilidade do código entre programadores. Essa convenção é muito seguida. Leia sobre ela pesquisando por “java 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 pedirá para modelar um “funcionário”. Existe um padrão para representar suas classes em diagramas, que é amplamente utilizado, chamado **UML**. Pesquise sobre ele.

4.12 – Exercícios: Orientação a objetos

O modelo de funcionários a seguir será utilizado para os exercícios de alguns dos posteriores capítulos. O objetivo aqui é criar um sistema para gerenciar os funcionários do Banco. Os exercícios desse capítulo são extremamente importantes.

1) Modele um funcionário. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário (`double`), a data de entrada no banco (`String`) e seu RG (`String`). Você deve criar alguns métodos de acordo com sua necessidade. Além deles, crie um método `recebeAumento` que aumenta o salário do funcionário de acordo com o parâmetro passado como argumento. Crie também um método `calculaGanhoAnual`, que não recebe parâmetro algum, devolvendo o valor do salário multiplicado por 12..

A ideia aqui é apenas modelar, isto é, só identifique que informações são importantes e o que um funcionário faz. Desenhe no papel tudo o que um `Funcionario` tem e tudo que ele faz.

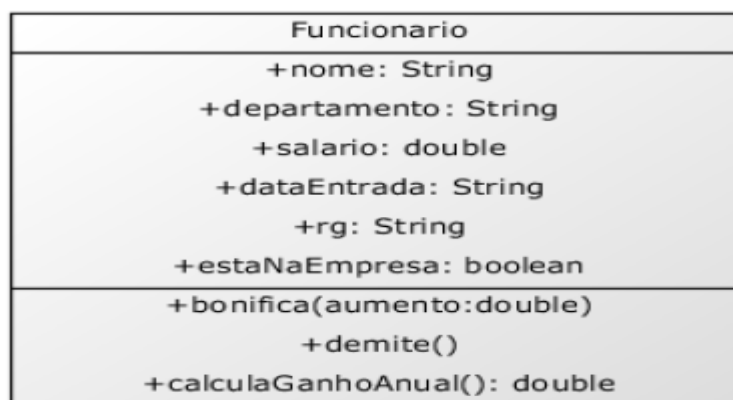
2) Transforme o modelo acima em uma classe Java. Teste-a, usando uma outra classe que tenha o `main`. Você deve criar a classe do funcionário com o nome `Funcionario`, mas pode nomear como quiser a classe de testes. A de teste deve possuir o método `main`.

Um esboço da classe:

```
class Funcionario {
    double salario;
    // seus outros atributos e métodos
    void recebeAumento(double aumento) {
        // o que fazer aqui dentro?
    }

    double calculaGanhoAnual() {
        // o que fazer aqui dentro?
    }
}
```

Você pode (e deve) compilar seu arquivo java sem que você ainda tenha terminado sua classe `Funcionario`. Isso evitará que você receba dezenas de erros de compilação de uma vez só. Crie a classe `Funcionario`, coloque seus atributos e, antes de colocar qualquer método, compile o arquivo java. O arquivo `Funcionario.class` será gerado, mas não podemos “executá-lo” já que essa classe não tem um `main`. De qualquer forma, a vantagem é que assim verificamos que nossa classe `Funcionario` já está tomando forma e está escrita em sintaxe correta.



Esse é um processo incremental. Procure desenvolver assim seus exercícios, para não descobrir só no fim do caminho que algo estava muito errado. Um esboço da classe que possui o main:

```
class TestaFuncionario {
    public static void main(String[] args) {
        Funcionario f1 = new Funcionario();
        f1.nome = "Hugo";
        f1.salario = 100;
        f1.recebeAumento(50);
        System.out.println("salario atual:" + f1.salario);
        System.out.println("ganho anual:" + f1.calculaGanhoAnual());
    }
}
```

Incremente essa classe. Faça outros testes, imprima outros atributos e invoque os métodos que você criou a mais.

Lembre-se de seguir a convenção java, isso é importantíssimo. Isto é, preste atenção nas maiúsculas e minúsculas, seguindo o seguinte exemplo: nomeDeAtributo, nomeDeMetodo, nomeDeVariavel, NomeDeClasse, etc...

Você até pode colocar todas as classes no mesmo arquivo e apenas compilar esse arquivo. Ele vai gerar um .class para cada classe presente nele. Porém, por uma questão de organização, é boa prática criar um arquivo .java para cada classe.

Em capítulos posteriores, veremos também determinados casos nos quais você será **obrigado** a declarar cada classe em um arquivo separado. Essa separação não é importante nesse momento do aprendizado, mas se quiser ir praticando sem ter que compilar classe por classe, você pode dizer para o javac compilar todos os arquivos java de uma vez:

```
javac *.java
```

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...
        // também pode imprimir this.calculaGanhoAnual()
    }
}
```

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

Para isso você vai precisar criar outra referência:

```
Funcionario f1 = new Funcionario();
f1.nome = "Danilo";
f1.salario = 100;

Funcionario f2 = new Funcionario();
f2.nome = "Danilo";
f2.salario = 100;

if (f1 == f2) {
    System.out.println("iguais");
} else {
    System.out.println("diferentes");
}
```

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();
f1.nome = "Hugo";
f1.salario = 100;
Funcionario f2 = f1;
```

O que acontece com o `if` do exercício anterior?

6) (opcional) 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, em que a Conta passou a ter referência para um Cliente).

```
class Funcionario {
    Data dataDeEntrada; // qual é o valor default aqui?
    // seus outros atributos e métodos
}
```

```
class Data {
    int dia;
    int mes;
    int ano;
}
```

Modifique sua classe TestaFuncionario para que você crie uma Data e atribua ela ao Funcionario:

```
Funcionario f1 = new Funcionario();
//...
Data data = new Data(); // ligação!
f1.dataDeEntrada = data;
```

Faça o desenho do estado da memória quando criarmos um Funcionario.

7) (opcional) Modifique seu método mostra para que ele imprima o valor da dataDeEntrada daquele Funcionario:

```
class Funcionario {
    // seus outros atributos e métodos
    Data dataDeEntrada;
    void mostra() {
        System.out.println("Nome: " + this.nome);
        // imprimir aqui os outros atributos...
        System.out.println("Dia: " + this.dataDeEntrada.dia);
        System.out.println("Mês: " + this.dataDeEntrada.mes);
        System.out.println("Ano: " + this.dataDeEntrada.ano);
    }
}
```

Teste-o. O que acontece se chamarmos o método mostra antes de atribuirmos uma data para este Funcionario?

8) (opcional) O que acontece se você tentar acessar um atributo diretamente na classe? Como, por exemplo:

```
Funcionario.salario = 1234;
```

Esse código faz sentido? E este:

```
Funcionario.calculaGanhoAtual();
```

Faz sentido perguntar para o esquema do Funcionario seu valor anual?

9) (opcional-avançado) Crie um método na classe Data que devolva o valor formatado da data, isto é, devolva uma String com “dia/mes/ano”. Isso para que o método mostra da classe Funcionario possa ficar assim:

```
class Funcionario {
    // atributos e metodos
    void mostra() {
        // imprime outros atributos...
        System.out.println("Data de entrada: " + this.dataDeEntrada.formatada());
    }
}
```

5 – Um pouco de arrays

5.1 – O problema

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

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

Isso pode se tornar um problema quando precisamos mudar a quantidade de variáveis a serem declaradas de acordo com um parâmetro. Esse parâmetro pode variar, como por exemplo, a quantidade de número contidos num bilhete de loteria. Um jogo simples possui 6 números, mas podemos comprar um bilhete mais caro, com 7 números ou mais.

Para facilitar esse tipo de caso podemos declarar um **vetor (array)** de inteiros:

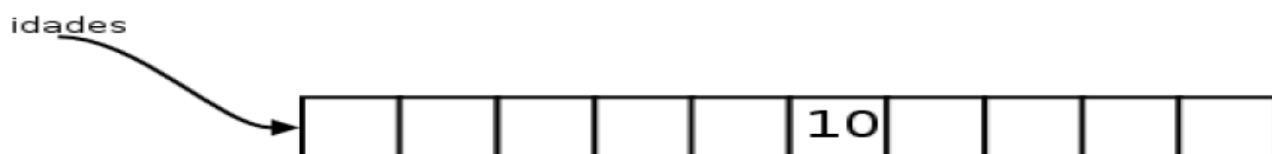
```
int[] idades;
```

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];
```

O que fizemos foi criar uma array de `int` de 10 posições e atribuir o endereço no qual ela foi criada. Podemos ainda 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 em que você criou o array. Se você tentar acessar uma posição fora desse alcance, um erro ocorrerá durante a execução.

Aprender a usar arrays 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, que tranquilo caso não consiga digerir toda sintaxe das arrays num primeiro momento.

No caso do bilhete de loteria, podemos utilizar o mesmo recurso. Mais ainda, a quantidade de números do nosso bilhete pode ser definido por uma variável. Considere que n indica quantos números nosso bilhete terá, podemos então fazer:

```
int numerosDoBilhete[] = new int[n];
```

E assim podemos acessar e modificar os inteiros com índice de 0 a n-1.

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, está 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:

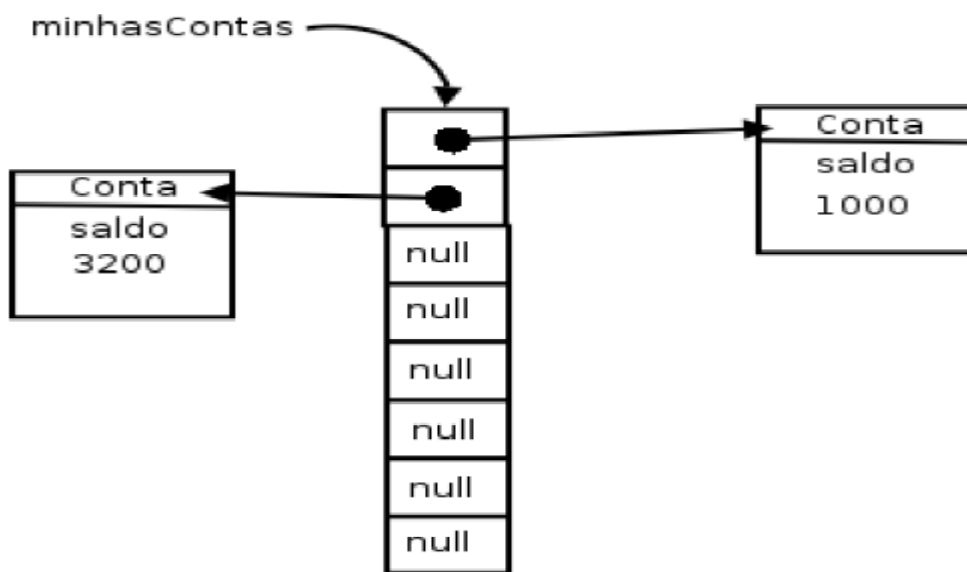
```
System.out.println(minhasContas[0].saldo);
```

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 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 um array

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

```
public static void main(String args[]) {
    int[] idades = new int[10];
    for (int i = 0; i < 10; i++) {
        idades[i] = i * 10;
    }
    for (int i = 0; i < 10; i++) {
        System.out.println(idades[i]);
    }
}
```

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

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

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 do array ao qual você está se referenciando naquele momento:

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

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 referir ela, copie os elementos da array velha.

5.4 – Percorrendo um array no Java 5.0

O Java 5.0 traz uma nova sintaxe para percorrermos 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 (que é uma grande parte dos casos), podemos usar o **enhanced-for**.

```
class AlgumaClasse{
    public static void main(String args[]) {
        int[] idades = new int[10];
        for (int i = 0; i < 10; i++) {
            idades[i] = i * 10;
        }

        // imprimindo toda a array
        for (int x : idades) {
            System.out.println(x);
        }
    }
}
```

Não precisamos mais do length para percorrer matrizes cujo tamanho não conhecemos:

```
class AlgumaClasse {
    void imprimeArray(int[] array) {
        for (int x : array) {
            System.out.println(x);
        }
    }
}
```

O mesmo é válido para arrays de referências. Esse for nada mais é que um truque de compilação para facilitar essa tarefa de percorrer arrays e torná-la mais legível.

5.5 – Exercícios: Arrays

1) Volte ao nosso sistema de Funcionario e crie uma classe Empresa dentro do mesmo arquivo .java. A Empresa tem um nome, cnpj e uma referência a uma array de Funcionario, além de outros atributos que você julgar necessário.

```
class Empresa {
```

```

    // outros atributos
    Funcionario[] empregados;
    String cnpj;
}

```

2) A Empresa deve ter um método adiciona, que recebe uma referência a Funcionario como argumento e guarda esse funcionário. Algo como:

```

void adiciona(Funcionario f) {
    // algo tipo:
    // this.empregados[ ??? ] = f;
    // mas que posição colocar?
}

```

Você deve inserir o Funcionario em uma posição da array que esteja livre. Existem várias maneiras 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 classe TestaEmpresa que possuirá um 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 empregados da classe Empresa não referencia lugar nenhum (seu valor é null):

```

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

```

Ou você pode construir a array dentro da própria declaração da classe Empresa, fazendo com que toda vez que uma Empresa é instanciada, a array de Funcionario que ela necessita também é criada. Crie alguns funcionários e passe como argumento para o adiciona da empresa:

```

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

```

```

Funcionario f2 = new Funcionario();
f2.salario = 1700;
empresa.adiciona(f2);

```

Você pode criar esses funcionários dentro de um loop e dar a cada um deles valores diferentes de salários:

```

for (int i = 0; i < 5; i++) {
    Funcionario f = new Funcionario();
    f.salario = 1000 + i * 100;
    empresa.adiciona(f);
}

```


Repare que temos de instanciar Funcionario dentro do laço. Se a instanciação de Funcionario ficasse acima do laço, estaríamos adicionado cinco vezes a **mesma** instância de Funcionario nesta Empresa e apenas mudando seu salário a cada iteração, que nesse caso não é o efeito desejado.

Opcional: o método adiciona pode gerar uma mensagem de erro indicando quando o array já está cheio.

4) Percorra o atributo empregados da sua instância da Empresa e imprima os salários de todos seus funcionários. Para fazer isso, você pode criar um método chamado mostraEmpregados dentro da classe Empresa:

```
void mostraEmpregados() {
    for (int i = 0; i < this.empregados.length; i++) {
        System.out.println("Funcionário na posição: " + i);
        // preencher para mostrar outras informacoes do funcionario
    }
}
```

Cuidado ao preencher esse método: alguns índices do seu array podem não conter referência para um Funcionario construído, isto é, ainda se referirem para null. Se preferir, use o for novo do java 5.0. Aí, através do seu main, depois de adicionar alguns funcionários, basta fazer:

```
empresa.mostraEmpregados();
```

5) (opcional) Em vez demonstrar apenas o salário de cada funcionário, você pode chamar o método mostra() de cada Funcionario da sua array.

6) (Opcional) Crie um método para verificar se um determinado Funcionario se encontra ou não como funcionário 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 ou o atributo livre.

7) (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.

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

6 – Modificadores de acesso e atributos de classe

6.1 – Controlando o acesso

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

```
class Conta {
    int numero;
    Cliente titular;
    double saldo;
    double limite;
    // ..

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

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

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

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:

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

Como evitar isso? Uma ideia 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 invocar o método `saca` e não permitir o acesso direto ao atributo. É o mesmo caso da validação de CPF.

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

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

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

Marcando um atributo como privado, fechamos o acesso ao mesmo em relação a todas as outras classes, fazendo com que o seguinte 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;
    }
}
```

TesteAcessoDireto.java:5 saldo has private access in Conta minhaConta.saldo = 1000;

^

1 error

Na orientação a objetos, é prática quase que obrigatória proteger seus atributos com `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 e facilitando futuras mudanças no sistema.

Muitas outras vezes nem mesmo queremos que outras classes saibam da existência de determinado atributo, escondendo-o por completo, já que ele diz respeito ao funcionamento interno do objeto.

Repare que, quem invoca o método `saca` não faz a menor ideia de que existe um limite que está sendo checado. Para quem for usar essa classe, basta saber o que o método faz e não como exatamente ele o faz (o que um método faz é sempre mais importante do que como ele faz: mudar a implementação é fácil, já mudar a assinatura de um método vai gerar problemas).

A palavra chave `private` também pode ser usada para modificar o acesso a um método. Tal funcionalidade é utilizada em diversos cenários: quando existe um método que serve apenas para auxiliar a própria classe e quando há código repetido dentro de dois métodos da classe são os mais comuns. Sempre devemos expor o mínimo possível de funcionalidades, para criar um baixo acoplamento entre as nossas classes.

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

```
class Conta {
    //...
    public void saca(double quantidade) {
        //posso sacar até saldo+limite
        if (quantidade > this.saldo + this.limite){
            System.out.println("Não posso sacar fora do limite!");
        } else {
            this.saldo = this.saldo - quantidade;
        }
    }
}
```

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 um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos. Algo muito mais educado que mexer diretamente em um atributo que não é seu!

Melhor ainda! O dia em que precisarmos mudar como é realizado um saque na nossa classe `Conta`, adivinhe onde precisaríamos modificar? Apenas no método `saca`, o que faz pleno sentido. Como exemplo, imagine cobrar CPMF de cada saque: basta você modificar ali, e nenhum outro código, fora a classe `Conta`, precisará ser recompilado. Mais: as classes que usam esse método nem precisam ficar sabendo de tal modificação! Você precisa apenas recompilar aquela classe e substituir aquele arquivo `.class`. Ganhamos muito em esconder o funcionamento do nosso método na hora de dar manutenção e fazer modificações.

6.2 – Encapsulamento

O que começamos a ver nesse capítulo é a ideia de **encapsular**, isto é, esconder todos os membros de uma classe (como vimos acima), além de esconder como funcionam as rotinas (no caso métodos) do nosso sistema.

Encapsular é **fundamental** para que seu sistema seja suscetível a mudanças: não precisaremos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está **encapsulada**. (veja o caso do método `saca`)



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

É sempre bom programar pensando na interface da sua classe, como seus usuários a estarão utilizando, e não somente em como ela vai 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. Sempre que vamos acessar um objeto, utilizamos sua interface. Existem diversas analogias fáceis no mundo real:

- Quando você dirige um carro, o que te importa são os pedais e o volante (interface) e não o motor que você está usando (implementação). É claro que um motor diferente pode te dar melhores resultados, mas **o que ele faz** é o mesmo que um motor menos potente, a diferença está em **como ele faz**. Para trocar um carro a álcool para um a gasolina você não precisa reaprender a dirigir! (trocar a implementação dos métodos não precisa mudar a interface, fazendo com que as outras classes continuem usando eles da mesma maneira).
- Todos os celulares fazem a mesma coisa (interface), eles possuem maneiras (métodos) de discar, ligar, desligar, atender, etc. O que muda é como eles fazem (implementação), mas repare que para o usuário comum pouco importa se o celular é GSM ou CDMA, isso fica encapsulado na implementação (que aqui são os circuitos).

Já temos conhecimentos suficientes para resolver 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 não seja válido
    }
    // ..
}
```

Se alguém tentar criar um Cliente e não usar o mudaCPF para alterar um cpf diretamente, vai receber um erro de compilação, já que o atributo CPF é **privado**. E o dia que você não precisar verificar o CPF de 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!

6.3 – Getters e Setters

O modificador private faz com que ninguém consiga modificar, nem mesmo ler, o atributo em questão. Com isso, temos um problema: como fazer para mostrar o saldo de uma Conta, já que nem mesmo podemos acessá-lo para leitura?

Precisamos então arranjar **uma maneira de** fazer esse acesso. Sempre que precisamos arrumar **uma maneira de fazer alguma coisa com um objeto**, utilizamos de métodos! Vamos então criar um método, digamos pegaSaldo, para realizar essa simples tarefa:

```
public class Conta {
    private double saldo;
    // outros atributos omitidos
    private double pegaSaldo() {
        return this.saldo;
    }
    // deposita() saca() e transfere() omitidos
}
```

Para acessarmos o saldo de uma conta, podemos fazer:

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

```

        Conta minhaConta = new Conta();
        minhaConta.deposita(1000);
        System.out.println("Saldo: " + minhaConta.pegarSaldo());
    }
}

```

Para permitir o acesso aos atributos (já que eles são `private`) de uma maneira controlada, a prática mais comum é criar dois métodos, um que retorna o valor e outro que muda o valor. A convenção para esses métodos é de colocar a palavra `get` ou `set` antes do nome do atributo. Por exemplo, a nossa conta com saldo, limite e titular fica assim, no caso da gente desejar dar acesso a leitura e escrita a todos os atributos:

```

public class Conta {
    private double saldo;
    private double limite;
    private Cliente titular;

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

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }

    public double getLimite() {
        return this.limite;
    }

    public void setLimite(double limite) {
        this.limite = limite;
    }

    public Cliente getTitular() {
        return this.titular;
    }

    public void setTitular(Cliente titular) {
        this.titular = titular;
    }
}

```

É uma má prática criar uma classe e, logo em seguida, criar getters e setters para todos 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 `deposita()` 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

precisássemos mudar como o saldo é mostrado. Podemos encapsular isso em um método e, porque não, dentro do próprio `getSaldo`? Repare:

```
public class Conta {  
    private double saldo;  
    private double limite;  
    private Cliente titular;  
    public double getSaldo() {  
        return this.saldo + this.limite;  
    }  
  
    // deposita() saca() e transfere() omitidos  
    public Cliente getTitular() {  
        return this.titular;  
    }  
  
    public void setTitular(Cliente titular) {  
        this.titular = titular;  
    }  
}
```

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 getters e setters não só ajuda 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 os objetos guardam seus dados. É uma prática muito importante.

Nossa classe está 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. Como já dito, não devemos criar getters e setters sem um motivo explícito.

6.4 – Construtores

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

```
class Conta {
    int numero;
    Cliente titular;
    double saldo;
    double limite;

    // construtor
    Conta() {
        System.out.println("Construindo uma conta.");
    }
    // ..
}
```

Então, quando fizermos:

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

A mensagem “construindo uma conta” aparecerá. É como uma rotina de inicialização que é chamada sempre que um novo objeto é criado. Um construtor pode parecer, mas **não é** um método. 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:

```
class Conta {
    int numero;
    Cliente titular;
    double saldo;
    double limite;

    // construtor
    Conta(Cliente titular) {
        this.titular = titular;
    }
}
```

```
// ..
}
```

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

```
Cliente carlos = new Cliente();
carlos.nome = "Carlos";
Conta c = new Conta(carlos);
System.out.println(c.titular.nome);
```

6.5 – A necessidade de um construtor

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

A ideia é bem simples. Se toda conta precisa de um titular, 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 a 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. 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.

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;
    Cliente titular;
    double saldo;
    double limite;

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

    Conta (int numero, Cliente titular) {
        this(titular); // chama o construtor que foi declarado acima
        this.numero = numero;
    }
}
```

```
}
```

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

No nosso exemplo do CPF, podemos forçar que a classe Cliente receba no mínimo o CPF, dessa maneira um Cliente já será construído e com um CPF válido.

Quando criamos uma classe com todos os atributos privados, seus getters e setters e um construtor vazio (padrão), na verdade estamos criando um Java Bean (mas não confunda com EJB, que é Enterprise Java Beans).

6.6 – Atributos de Classe

Nosso banco também quer controlar a quantidade de contas existentes no sistema. Como poderíamos fazer isto? A ideia 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.** 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, mas 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 conta antes de chamar o método! Isso não é legal, pois gostaríamos de saber quantas contas existem sem precisar ter acesso a um objeto conta. A ideia aqui é a mesma, transformar esse método que todo objeto conta tem em 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 uma Conta, e sim usando o nome da classe.

Métodos e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma classe, o que faz todo sentido já que dentro de um método estático não temos acesso à referência this, pois um método estático é chamado através da classe, e não de um objeto.

O static realmente traz um “cheiro” procedural, porém em muitas vezes é necessário.

6.7 – Exercícios: Encapsulamento, Construtores e Static

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

2) Crie os getters e setters necessários da sua classe Funcionario. Por exemplo:

```
class Funcionario {
    private double salario;
    // ...
    public double getSalario() {
        return this.salario;
    }
    public void setSalario(double salario) {
        this.salario = salario;
    }
}
```

Não copie e cole! Aproveite para praticar sintaxe. Logo passaremos a usar o Eclipse e aí sim teremos procedimentos mais simples para este tipo de tarefa. Repare que o método calculaGanhoAnual parece também um getter. Aliás, seria comum alguém nomeá-lo de getGanhoAnual. Getters não precisam apenas retornar atributos. Eles podem trabalhar com esses dados.

3) Modifique suas classes que acessam e modificam atributos de um Funcionario para utilizar os getters e setters recém criados.

Por exemplo, onde você encontra:

```
f.salario = 100;
System.out.println(f.salario);
```

passa para:

```
f.setSalario(100);
System.out.println(f.getSalario());
```

4) Faça com que sua classe Funcionario possa receber, opcionalmente, o nome do Funcionario durante a criação do objeto. Utilize construtores para obter esse resultado.

Dica: utilize um construtor sem argumentos também, para o caso de a pessoa não querer passar o nome do Funcionario. Seria algo como:

```
class Funcionario {
    public Funcionario() {
        // construtor sem argumentos
    }
    public Funcionario(String nome) {
        // construtor que recebe o nome
    }
}
```

Por que você precisa do construtor sem argumentos para que a passagem do nome seja opcional?

5) (opcional) 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?

6) (opcional) Crie os getters e setters da sua classe Empresa e coloque seus atributos como private. 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 funcionários?

Não seria mais interessante ter um método como este?

```
class Empresa {
    // ...
    public Funcionario getFuncionario (int posicao) {
        return this.empregados[posicao];
    }
}
```

7) (opcional) Na classe Empresa, em vez de criar um array de tamanho fixo, receba como parâmetro o construtor o tamanho do array de Funcionario. Com esse construtor, o que acontece se tentarmos dar new Empresa() sem passar argumento algum? Por quê?

8) (opcional) Como garantir que datas como 31/2/2012 não sejam aceitas pela sua classe Data?

9) (opcional) Crie a classe PessoaFisica. Queremos ter a garantia de que pessoa física 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 – Herança, reescrita e polimorfismo

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, além do número de funcionários que ele gerencia:

```
class Gerente {
    String nome;
    String cpf;
    double salario;
    int senha;
    int numeroDeFuncionariosGerenciados;

    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
}
```

Poderíamos ter deixado a classe Funcionario mais genérica, mantendo nela senha de acesso, e o número de funcionários gerenciados. Caso o funcionário não fosse um gerente, deixaríamos estes atributos vazios. Essa é uma possibilidade, porém podemos começar a ter muito atributos opcionais, e a classe ficaria estranha. E em relação aos métodos? A classe Gerente tem o método autentica, que não faz sentido existir em um funcionário que não é gerente.

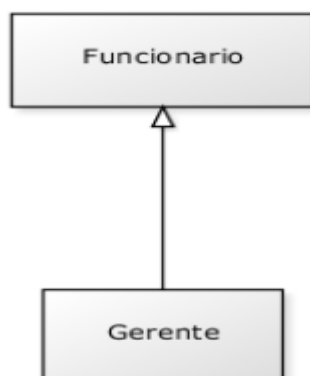
Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente! Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizarmos as informações principais do funcionário em um único lugar!

Existe um jeito, 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.

```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;
    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }
    // setter da senha omitido
}
```

```
}
```

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



```

class TestaGerente {
    public static void main(String[] args) {
        Gerente gerente = new Gerente();
        // podemos chamar métodos do Funcionario:
        gerente.setNome("João da Silva");
        // e também métodos do Gerente!
        gerente.setSenha(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 acessá-los diretamente. Para acessar um membro privado na filha indiretamente, seria necessário que a mãe expusesse um outro método visível que invocasse esse atributo ou método privado.

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. Outra forma é dizer que Funcionario é classe **mãe** de Gerente e Gerente é classe **filha** de Funcionario.

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 e por suas subclasses (e mais algumas outras classes, mas veremos isso em outro capítulo).

```

class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;
    // métodos devem vir aqui
}

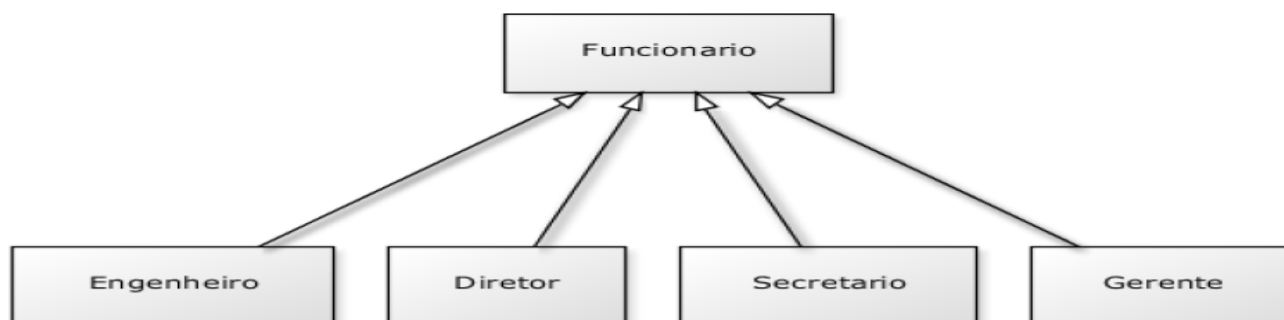
```

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

atributos. Essa é uma discussão um pouco mais avançada. Além disso, não só as subclasses, mas também as outras classes, podem acessar os atributos `protected`, que veremos mais a frente (mesmo pacote).

Da mesma maneira, podemos ter uma classe `Diretor` que estenda `Gerente` e a classe `Presidente` pode estender diretamente de `Funcionario`. Fique claro que essa é uma decisão de negócio. Se `Diretor` vai estender de `Gerente` ou não, vai depender se, para você, `Diretor` é um `Gerente`.

Uma classe pode ter várias filhas, mas pode ter apenas uma mãe, é a chamada herança simples do java.



7.2 – Reescrita do 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());
  
```

O resultado aqui será 500. Não queremos essa resposta, pois o gerente deveria ter 750 de bônus nesse caso. Para consertar isso, uma das opções seria criar um novo método na classe `Gerente`, chamado, por exemplo, `getBonificacaoDoGerente`. O problema é que teríamos dois métodos em `Gerente`, confundindo bastante quem for usar essa classe, além de que cada um da uma resposta diferente.

No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos **reescrever** (reescrever, sobrescrever, `override`) este método:

```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;
    public double getBonificacao() {
        return this.salario * 0.15;
    }
    // ...
}
```

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

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

7.3 – Invocando o método reescrito

Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da classe mãe: realmente alteramos o seu comportamento. Mas podemos invocá-lo no caso de estarmos dentro da classe.

Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo de um Funcionario porém adicionando R\$ 1000. Poderíamos fazer assim:

```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;
    public double getBonificacao() {
        return this.salario * 0.10 + 1000;
    }
    // ...
}
```

Aqui teríamos um problema: o dia que o getBonificacao do Funcionario mudar, precisaremos mudar o método do Gerente para acompanhar a nova bonificação. Para evitar isso, o getBonificacao do Gerente pode chamar o do Funcionario utilizando a palavra chave super.

```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;
    public double getBonificacao() {
        return super.getBonificacao() + 1000;
    }
}
```

Essa invocação vai procurar o método com o nome getBonificacao de uma super classe de Gerente. No caso ele logo vai encontrar esse método em Funcionario. Essa é uma prática comum, pois muitos casos o método reescrito geralmente faz “algo a mais” que o método da classe mãe.

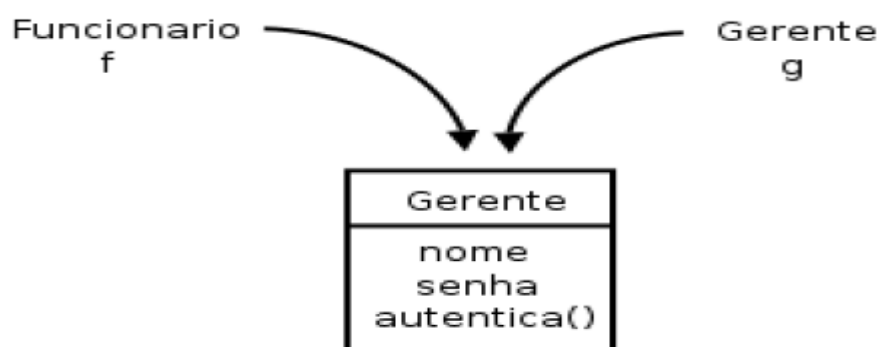
Chamar ou não o método de cima é uma decisão sua e depende do seu problema. Algumas vezes não faz sentido invocar o método que reescrevemos.

7.4 - Polimorfismo

O que guarda uma variável do tipo Funcionario? Uma referência para um Funcionario, nunca o objeto em si.

Na herança, vimos que todo Gerente é um Funcionario, pois é uma extensão deste. Podemos nos referir a um Gerente como sendo um Funcionario. Se alguém precisa falar com um Funcionario do banco, pode falar com um Gerente! Porque? Pois Gerente **é um** Funcionario. Essa é a semântica da herança.

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.setSalario(5000.0);
```



Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos 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 invocação de método sempre vai ser **decidida em tempo de execução**. O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não coma que estamos usando para referenciá-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. Por que faríamos 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 registra(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.registra(funcionario1);

```

```

Funcionario funcionario2 = new Funcionario();
funcionario2.setSalario(1000.0);
controle.registra(funcionario2);
System.out.println(controle.getTotalDeBonificacoes());

```

Repare que conseguimos passar um Gerente para um método que recebe um Funcionario como argumento. Pense como numa porta na agência bancária com o seguinte aviso: “Permitida a entrada apenas de Funcionários”. Um gerente pode passar nessa porta? Sim, pois Gerente **é um** Funcionario.

Qual será o valor resultante? Não importa que dentro do método registra do ControleDeBonificacoes receba Funcionario. Quando ele receber um objeto que realmente é um Gerente, o seu método reescrito será invocado. Reafirmando: **não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele.**

No dia em que criarmos uma classe Secretaria, por exemplo, que é filha de Funcionario, precisaremos mudar a classe de ControleDeBonificacoes? Não. Basta a classe Secretaria reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, juntamente com a reescrita de método: diminuir o acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou ControleDeBonificacoes pode nunca ter imaginado a criação da classe Secretaria ou Engenheiro. Contudo, não será necessário reimplementar esse controle em cada nova classe: reaproveitamos aquele código.

Note que o uso de herança **aumenta** o acoplamento entre as classes, isto é, o quanto uma classe depende de outra. A relação entre classe mãe e filha é muito forte e isso acaba fazendo com que o programador das classes filhas tenha que conhecer a implementação da classe pai e vice-versa, fica difícil fazer uma mudança pontual no sistema.

Por exemplo, imagine se tivermos que mudar algo na nossa classe Funcionario, mas não quiséssemos que todos os funcionários sofressem a mesma mudança. Precisaríamos passar por cada uma das filhas de Funcionario verificando se ela se comporta como deveria ou se devemos sobrescrever o tal método modificado. Esse é um problema da herança, e não do polimorfismo, que resolveremos mais tarde com a ajuda de Interfaces.

7.5 – Um outro exemplo

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 que forem necessários
}
```

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(EmpregadoDaFaculdade f) {
        System.out.println(f.getInfo());
        System.out.println(f.getGastos());
    }
}
```

Podemos passar para nossa classe qualquer `EmpregadoDaFaculdade`! 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 `EmpregadoDaFaculdade`, será que vamos precisar alterar algo na nossa classe de Relatório? Não. Essa é a ideia! Quem programou a classe `GeradorDeRelatorio` nunca imaginou que existiria uma classe `Reitor` e, mesmo assim, o sistema funciona.

```
class Reitor extends EmpregadoDaFaculdade {
    // informações extras
    String getInfo() {
        return super.getInfo() + " e ele é um reitor";
    }
    // não sobrescrevemos o getGastos!!!
}
```

7.6 – Exercícios: Herança e Polimorfismo

1) Vamos criar uma classe `Conta`, que possua um saldo os métodos para pegar saldo, depositar e sacar.

a) Crie a classe `Conta`:

```
public class Conta {
}
```

b) Adicione o atributo `saldo`

```
public class Conta {
    private double saldo;
}
```

c) Crie os métodos `getSaldo()`, `deposita(double)` e `saca(double)`

```
public class Conta {
    private double saldo;
    public void deposita(double valor) {
        this.saldo += valor;
    }
    public void saca(double valor) {
        this.saldo -= valor;
    }
    public double getSaldo() {
        return this.saldo;
    }
}
```

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

```
class Conta {
    private double saldo;
    // outros métodos aqui também ...
}
```

```

        void atualiza(double taxa) {
            this.saldo += this.saldo * taxa;
        }
    }

```

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 taxa e a ContaPoupanca deve atualizar-se com o triplo da taxa. Além disso, a ContaCorrente deve reescrever o método deposita, afim de retirar uma taxa bancária de dez centavos de cada depósito.

- Crie as classes ContaCorrente e ContaPoupanca. Ambas são filhas da classe Conta:

```

public class ContaCorrente extends Conta {
}

```

```

public class ContaPoupanca extends Conta {
}

```

- Reescreva o método atualiza na classe ContaCorrente, seguindo o enunciado:

```

public class ContaCorrente extends Conta {
    public void atualiza(double taxa) {
        this.saldo += this.saldo * taxa * 2;
    }
}

```

Repare que, para acessar o atributo saldo herdado da classe Conta, **você vai precisar trocar o modificador de visibilidade de saldo para protected.**

- Reescreva o método atualiza na classe ContaPoupanca, seguindo o enunciado:

```

public class ContaPoupanca extends Conta {
    public void atualiza(double taxa) {
        this.saldo += this.saldo * taxa * 3;
    }
}

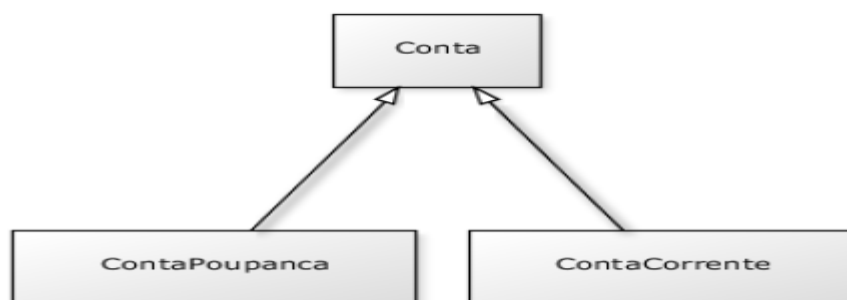
```

- Na classe ContaCorrente, reescreva o método deposita para descontar a taxa bancária de dez centavos:

```

public class ContaCorrente extends Conta {
    public void atualiza(double taxa) {
        this.saldo += this.saldo * taxa * 2;
    }
    public void deposita(double valor) {
        this.saldo += valor - 0.10;
    }
}

```



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

```

public class TestaContas {
    public static void main(String[] args) {
        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);
        System.out.println(c.getSaldo());
        System.out.println(cc.getSaldo());
        System.out.println(cp.getSaldo());
    }
}
  
```

Após imprimir 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? Realmente, essa não é a maneira mais útil do polimorfismo - veremos o seu real poder no próximo exercício. Porém existe uma utilidade de declararmos uma variável de um tipo menos específico do que o objeto realmente é. É **extremamente importante** perceber que não importa como nos referimos a um objeto, o método que será invocado é sempre o mesmo! A JVM vai descobrir em tempo de execução qual deve ser invocado, dependendo de que tipo é aquele objeto, não importando como nos referimos a ele.

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. Além disso, conforme atualiza as contas, o banco quer saber quanto do dinheiro do banco foi atualizado até o momento. Por isso, precisamos ir guardando o saldoTotal e adicionar um getter à classe.


```

public class AtualizadorDeContas {
    private double saldoTotal = 0;
    private double selic;
    public AtualizadorDeContas(double selic) {
        this.selic = selic;
    }
    public void roda(Conta c) {
        // aqui você imprime o saldo anterior, atualiza a conta,
        // e depois imprime o saldo final
        // lembrando de somar o saldo final ao atributo saldoTotal
    }
    // outros métodos, colocar o getter para saldoTotal!
}

```

7) (opcional) No método main, vamos criar algumas contas e rodá-las:

```

public class TestaAtualizadorDeContas {
    public static void main(String[] args) {
        Conta c = new Conta();
        Conta cc = new ContaCorrente();
        Conta cp = new ContaPoupanca();
        c.deposita(1000);
        cc.deposita(1000);
        cp.deposita(1000);
        AtualizadorDeContas adc = new AtualizadorDeContas(0.01);
        adc.roda(c);
        adc.roda(cc);
        adc.roda(cp);
        System.out.println("Saldo Total: " + adc.getSaldoTotal());
    }
}

```

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

9) (Opcional) Se você precisasse criar uma classe ContaInvestimento, e seu método atualiza fosse complicadíssimo, você precisaria alterar a classe AtualizadorDeContas?

10) (Opcional, Trabalhoso) Crie uma classe Banco que possui um array de Conta. Repare que num array de Conta você pode colocar tanto ContaCorrente quanto ContaPoupanca. Crie um método public void adiciona(Conta c), um método public Conta pegaConta(int x) e outro public 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.

8 – Eclipse IDE

8.1 – O Eclipse

O Eclipse (<http://www.eclipse.org>) é uma IDE (integrated development environment). Diferente de uma RAD, onde o objetivo é desenvolver o mais rápido possível através do arrastar-e-soltar do mouse, onde montanhas de código são gerados em background, uma IDE te auxilia no desenvolvimento, evitando se intrometer e fazer muita magia.

O Eclipse é a IDE líder de mercado. Formada por um consórcio liderado pela IBM, possui seu código livre. A última versão é a 4.2, mas com qualquer versão posterior a do 3.1 você terá suporte ao Java 5, 6 e 7.

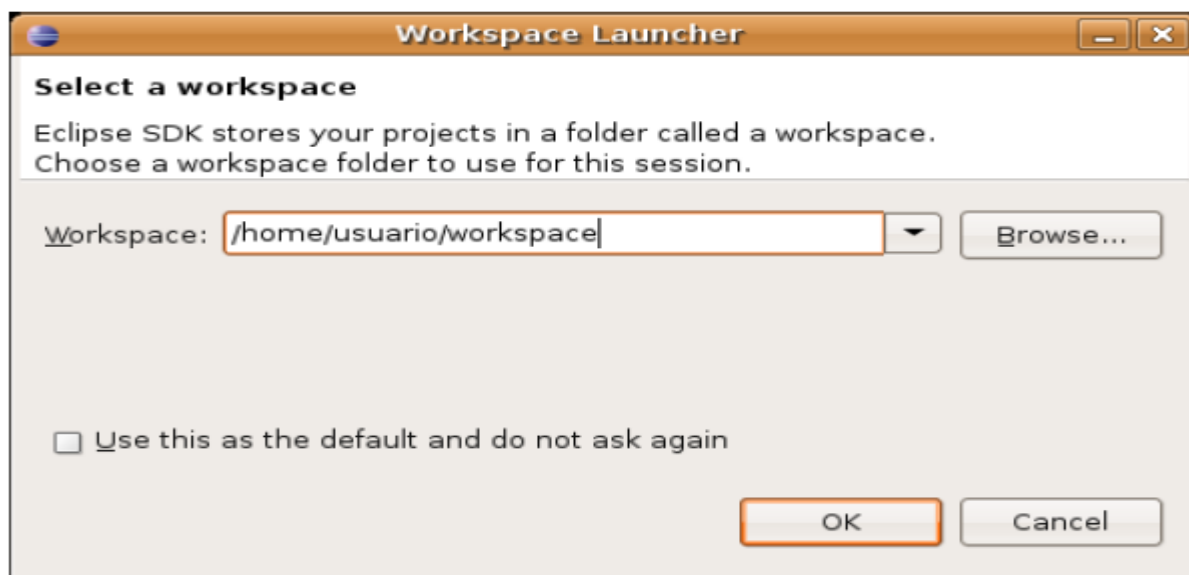
Veremos aqui os principais recursos do Eclipse. Você perceberá que ele evita ao máximo te atrapalhar e apenas gera trechos de códigos óbvios, sempre ao seu comando. Existem também centenas de plugins gratuitos para gerar diagramas UML, suporte a servidores de aplicação, visualizadores de banco de dados e muitos outros.

Baixe o Eclipse do site oficial <http://www.eclipse.org>. Apesar de ser escrito em Java, a biblioteca gráfica usada no Eclipse, chamada SWT, usa componentes nativos do sistema operacional. Por isso você deve baixar a versão correspondente ao seu sistema operacional. Descompacte o arquivo e pronto, basta rodar o executável.

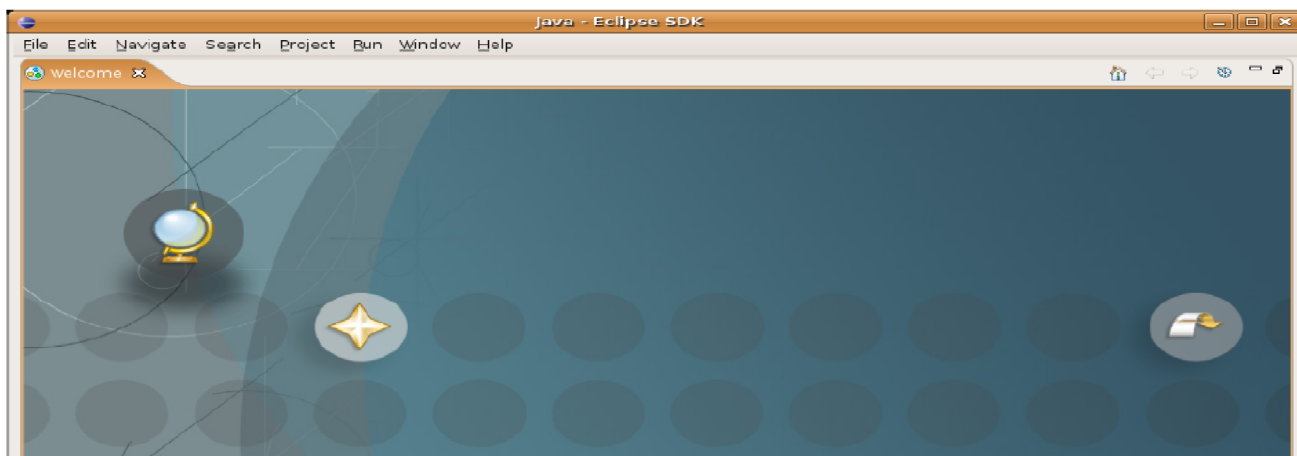
Uma outra IDE open source famosa é o Netbeans, da Oracle. (<http://www.netbeans.org>). Além dessas, Oracle, Borland e a própria IBM possuem IDEs comerciais e algumas versões mais restritas de uso livre. A empresa JetBrains desenvolve o IntelliJ IDEA, uma IDE paga que tem ganho muitos adeptos.

8.2 – Apresentando o Eclipse

Clique no ícone do Eclipse no seu Desktop. A primeira pergunta que ele te faz é que workspace você vai usar. Workspace define o diretório em que as suas configurações pessoais e seus projetos serão gravados.

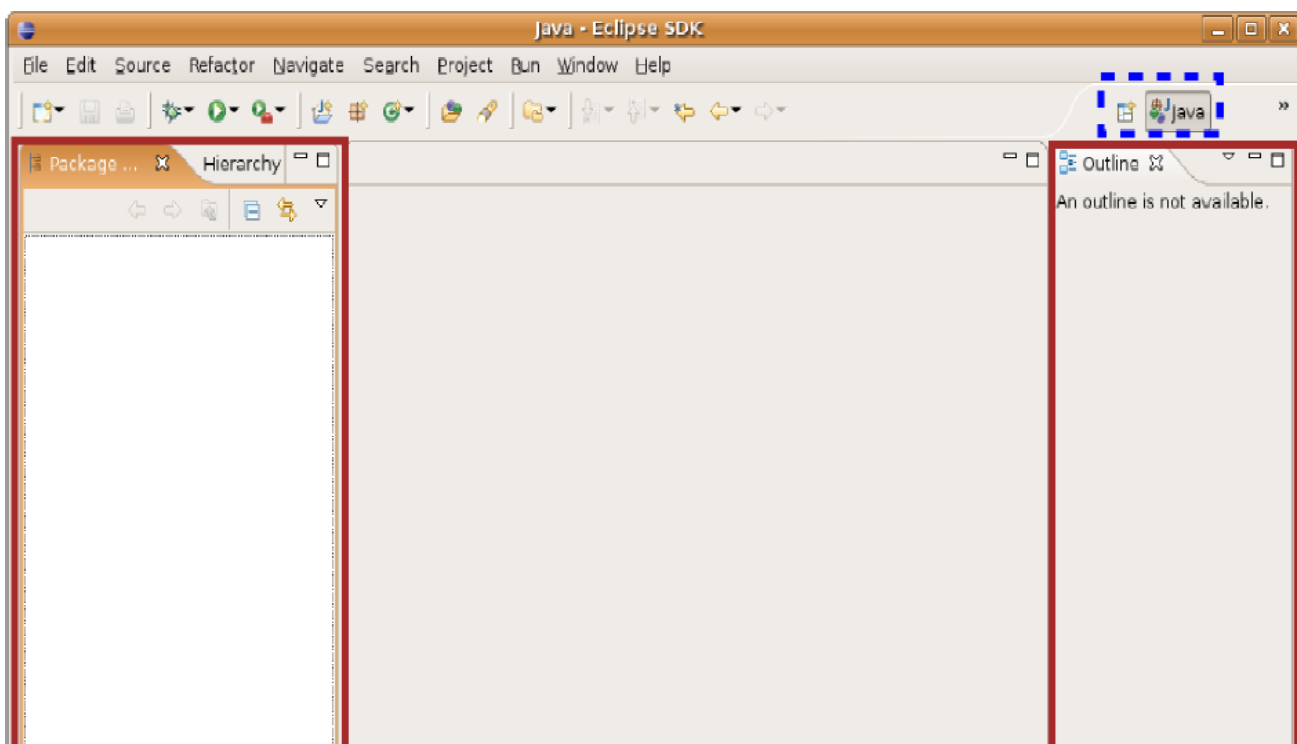


Você pode deixar o diretório pré-definido. Logo em seguida, uma tela de Welcome será aberta, onde você tem diversos links para tutoriais e ajuda. Clique em Workbench. A tela de Welcome do Eclipse 4 é um pouco diferente, mas possui exatamente os mesmos recursos nos mesmos locais.

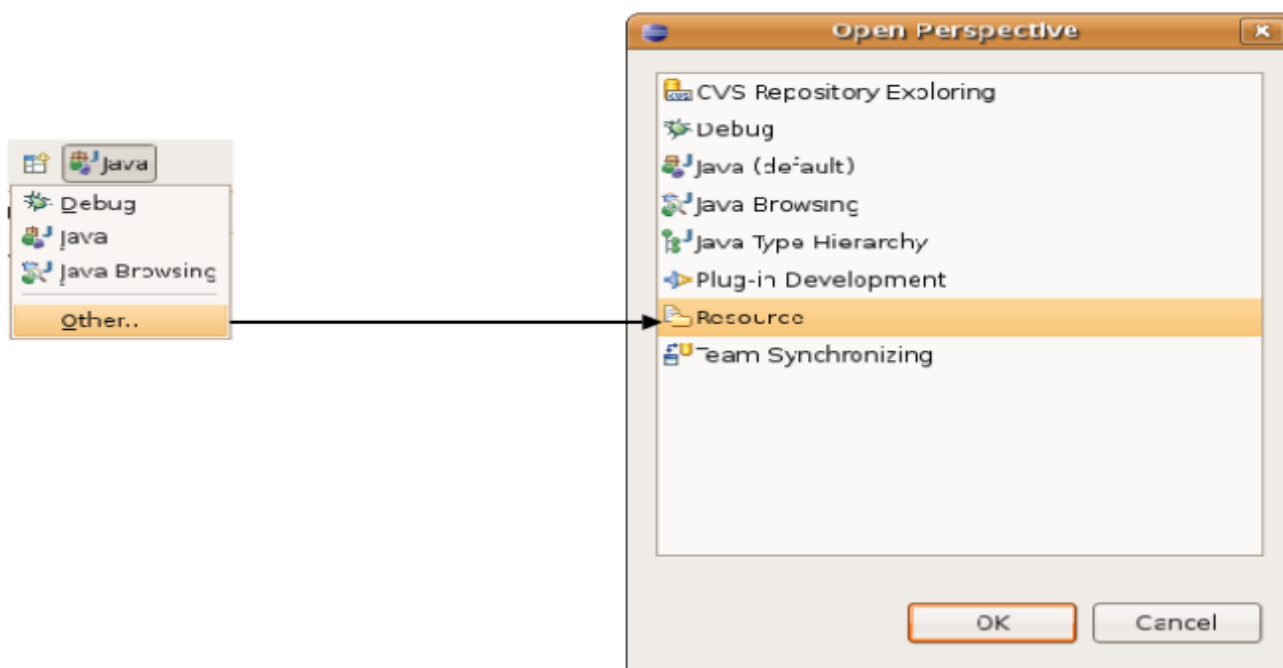


8.3 – Views e Perspective

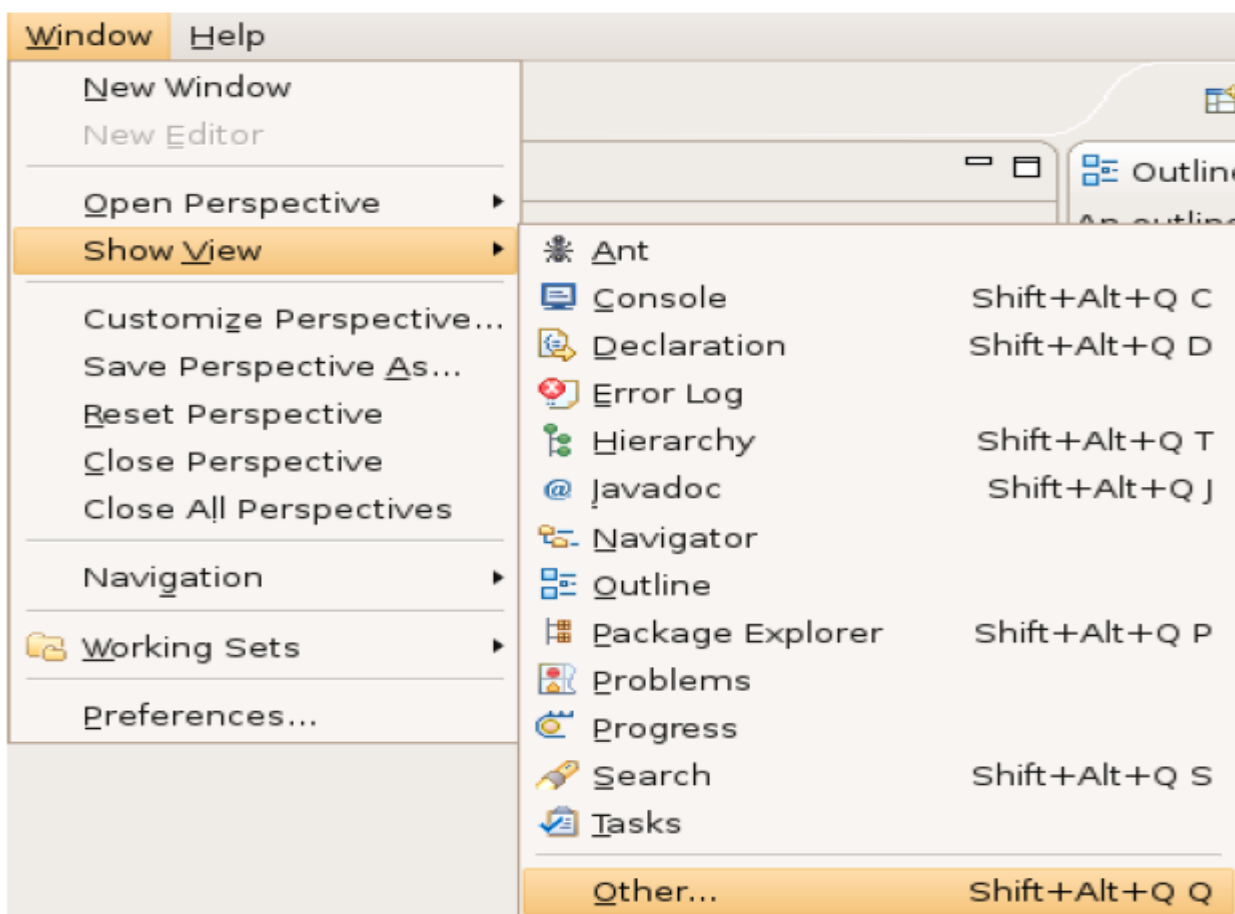
Feche a tela de Welcome e você verá a tela abaixo. Nesta tela, destacamos as Views (em linha contínua) e as Perspectives (em linha pontilhada) do Eclipse.



Mude para a perspectiva Resource, clicando no ícone ao lado da perspectiva Java, selecionando Other e depois Resource. Neste momento, trabalharemos com esta perspectiva, antes da de Java, pois ela possui um conjunto de Views mais simples.

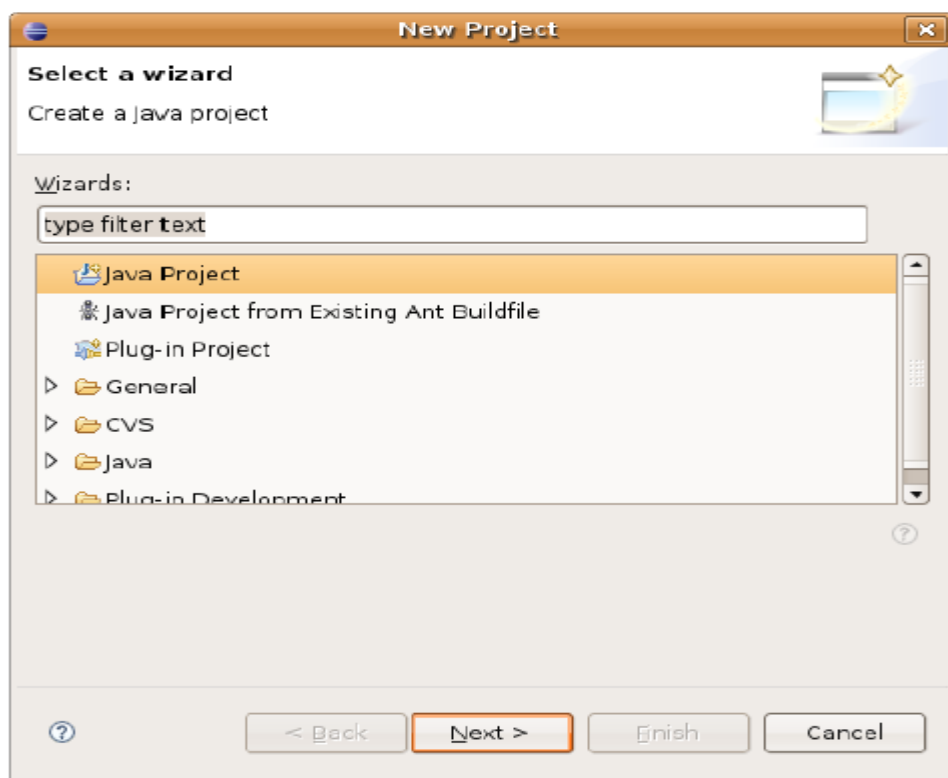
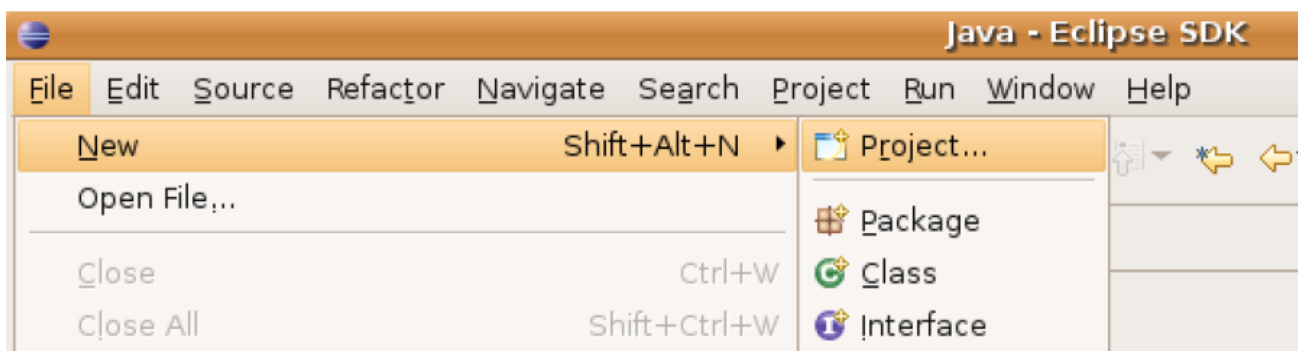


A View Navigator mostra a estrutura de diretório assim como está no sistema de arquivos. A View Outline mostra um resumo das classes, interfaces e enumerações declaradas no arquivo java atualmente editado (serve também para outros tipos de arquivos). No menu **Window -> Show View -> Other**, você pode ver as dezenas de Views que já vem embutidas no Eclipse. Acostume-se a sempre procurar novas Views, elas podem te ajudar em diversas tarefas.

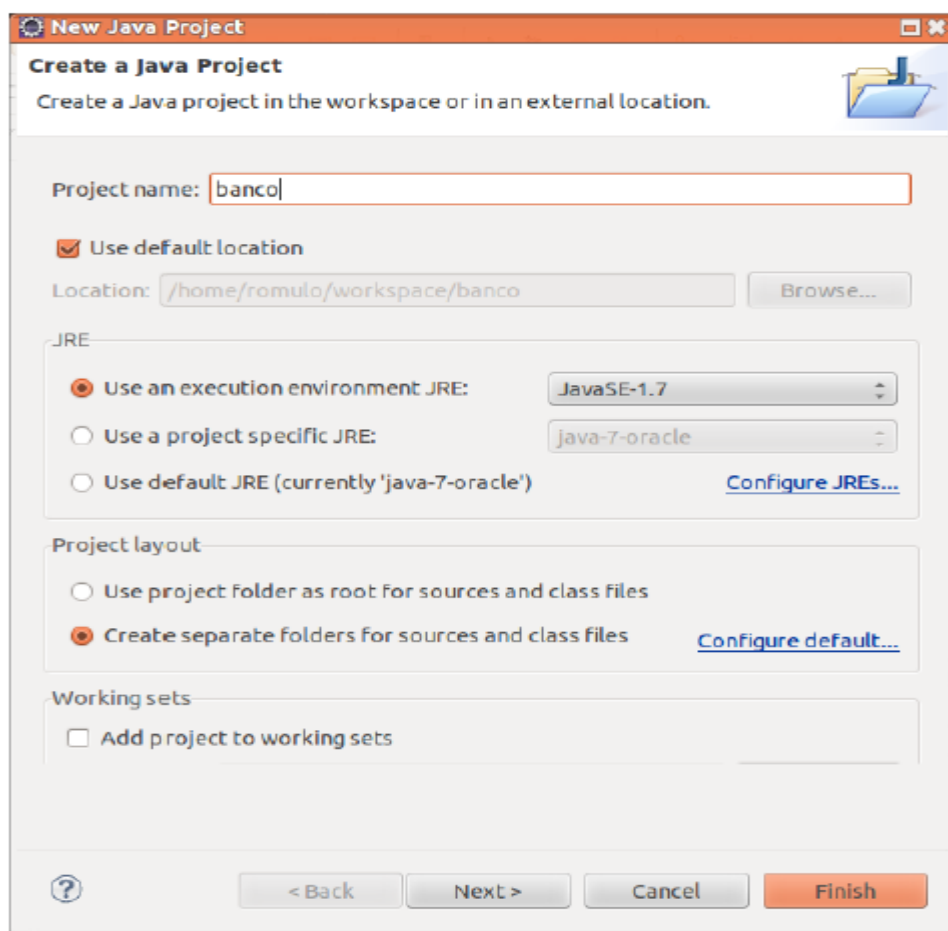


8.4 – Criando um novo projeto

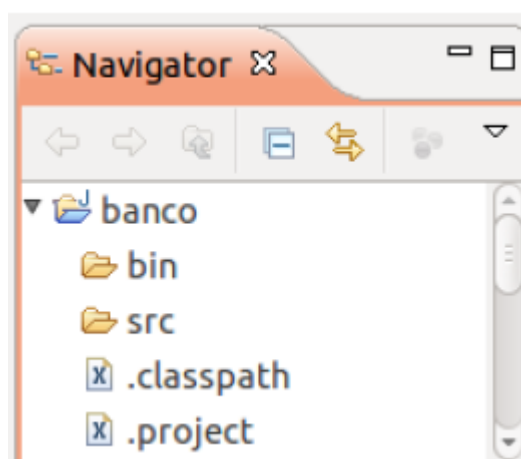
Vá em **File -> New -> Project**. Selecciona Java Project e clique em Next.



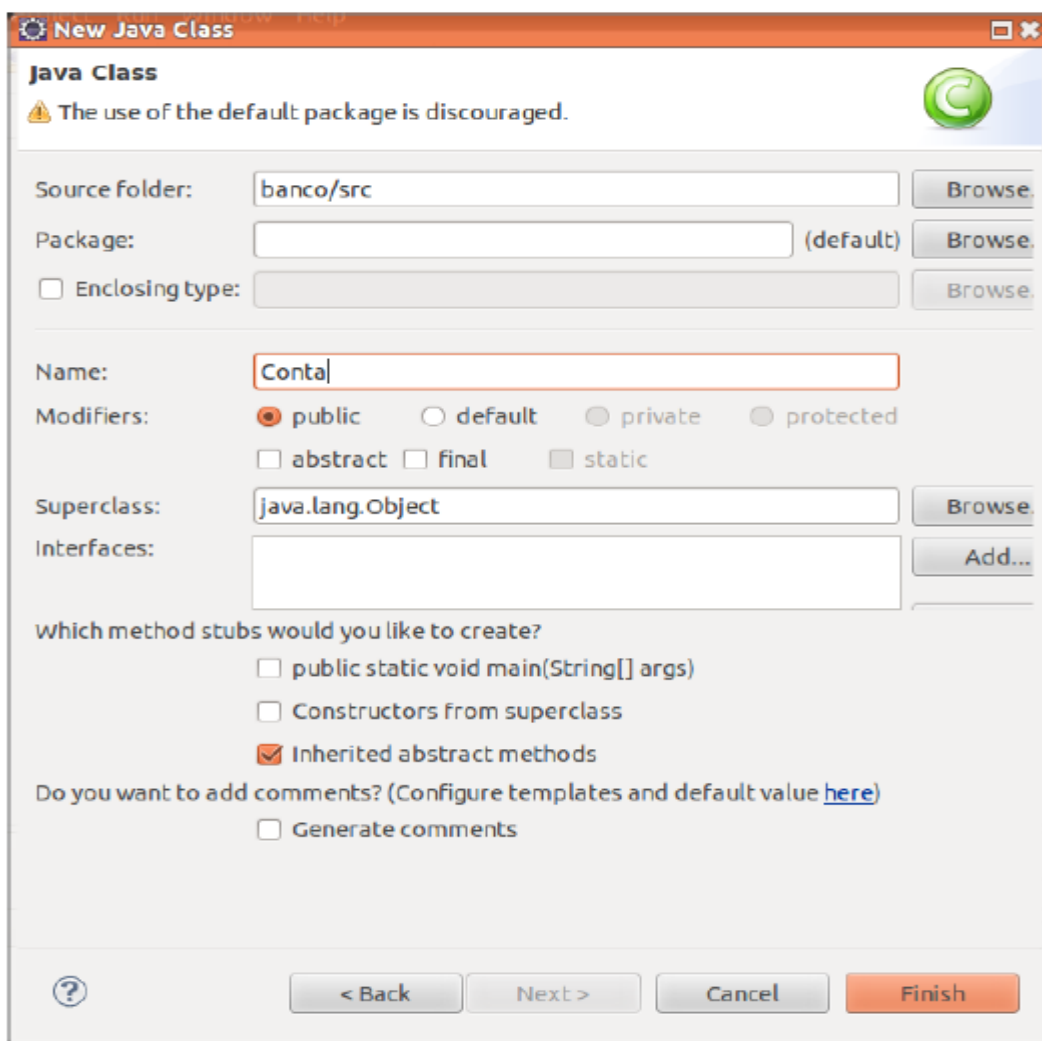
Crie um projeto chamado banco. Você pode chegar nessa mesma tela clicando com o botão da direita no espaço da View Navigator e seguindo o mesmo menu. Nesta tela, configure seu projeto como na tela abaixo:



Isto é, marque “create separate source and output folders”, desta maneira seus arquivos java e arquivos class estarão em diretórios diferentes, para você trabalhar de uma maneira mais organizada. Clique em Finish. O Eclipse pedirá para trocar a perspectiva para Java; escolha “No” para Permanecer em Resource. Na View Navigator, você verá o novo projeto e suas pastas e arquivos:

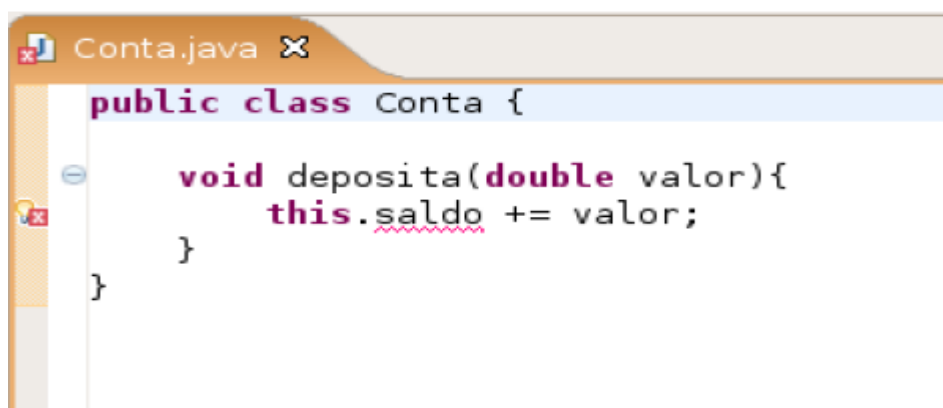


Vamos iniciar nosso projeto criando a classe Conta. Para isso, vá em File -> New -> Other -> Class. Clique em Next e crie a classe seguindo a tela abaixo:

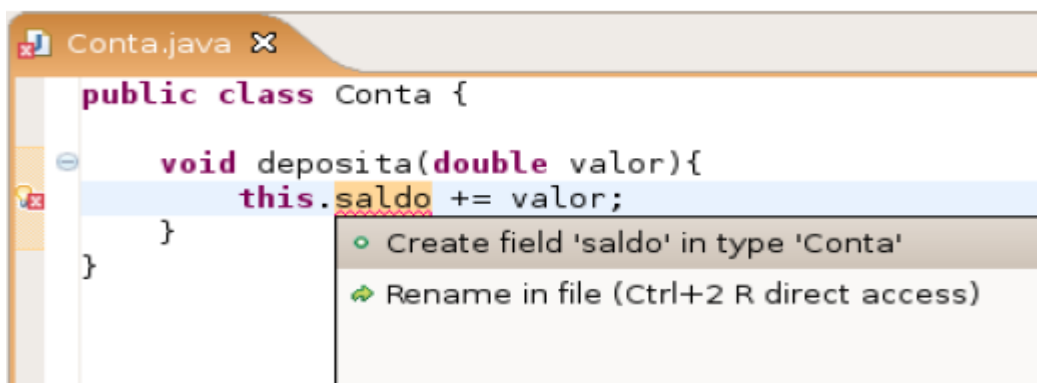


Clique em Finish. O Eclipse possui diversos wizards, mas usaremos o mínimo deles. O interessante é usar o code assist e quick fixes que a ferramenta possui e veremos em seguida. Não se atente às milhares de opções de cada wizard, a parte mais interessante do Eclipse não é essa.

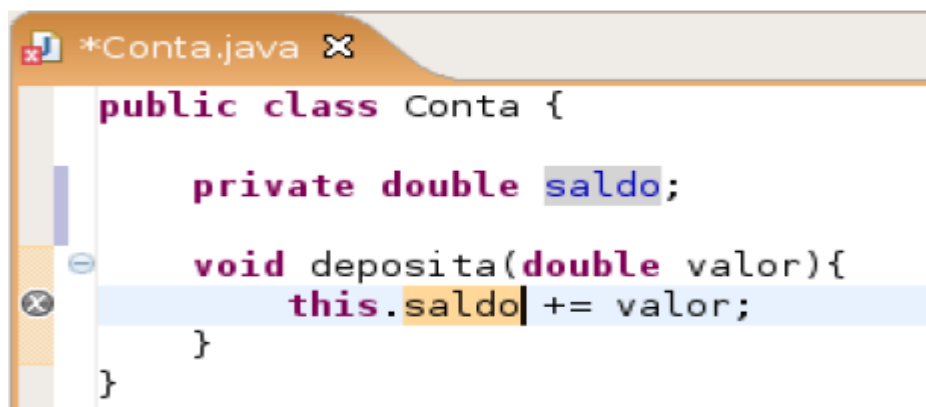
Escreva o método deposita como abaixo e note que o Eclipse reclama de erro em `this.saldo` pois este atributo não existe.



Vamos usar o recurso do Eclipse de **quick fix**. Coloque o cursor em cima do erro e aperte Ctrl + 1.

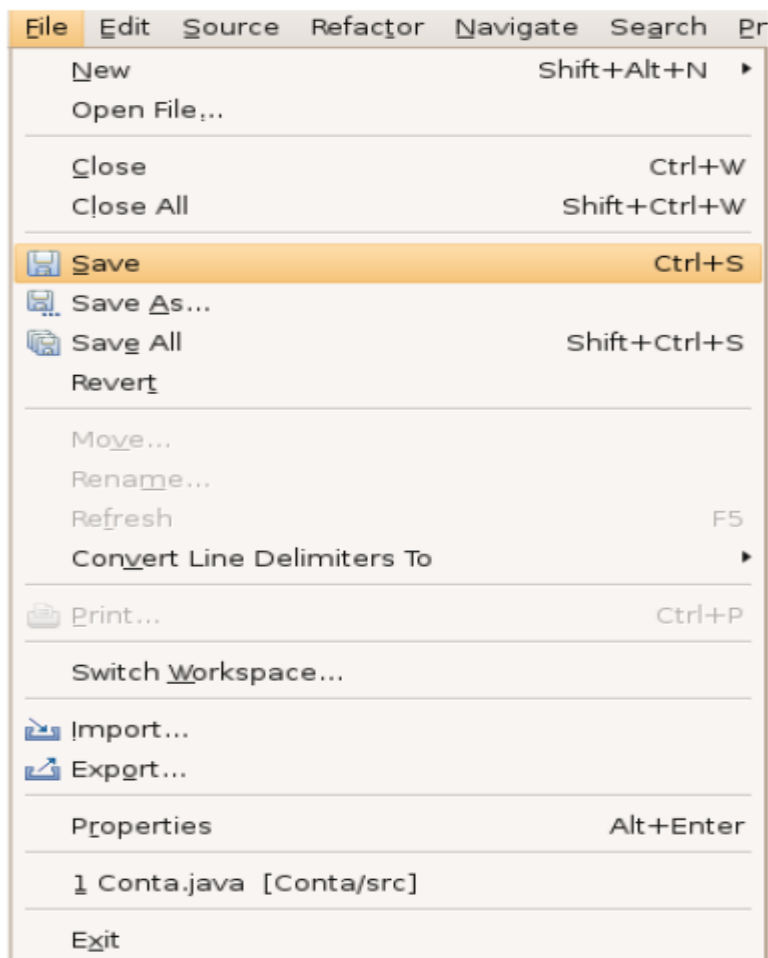


O Eclipse sugerirá possíveis formas de consertar o erro; uma delas é, justamente, criar o campo saldo na classe Conta, que é nosso objetivo. Clique nesta opção.



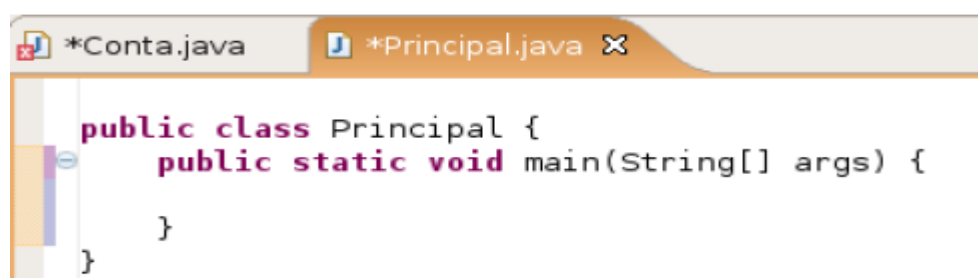
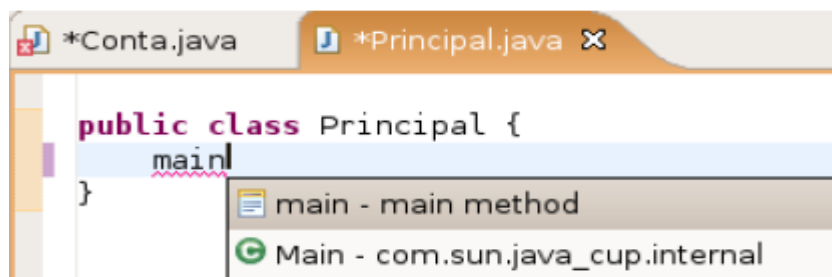
Este recurso de quick fixes, acessível pelo Ctrl+1, é uma das grandes facilidades do Eclipse e é extremamente poderoso. Através dele é possível corrigir boa parte dos erros na hora de programar e, como fizemos, economizar a digitação de certos códigos repetitivos. No nosso exemplo, não precisamos criar o campo antes; o Eclipse faz isso para nós. Ele até acerta a tipagem, já que estamos somando ele a um double. O private é colocado por motivos que já estudamos.

Vá ao menu File -> Save para gravar. Control + S tem o mesmo efeito.



8.5 – Criando o MAIN

Crie uma nova classe chamada Principal. Vamos colocar um método main para testar nossa Conta. Em vez de digitar todo o método main, vamos usar o **code assist** do Eclipse. Escreva só main e aperte Ctrl + Espaço logo em seguida.



O Eclipse sugerirá a criação do método main completo; selecione esta opção. O control + espaço é chamado de **code assist**. Assim como os quick fixes são de extrema importância. Experimente usar o code assist em diversos lugares.

Dentro do método main, comece a digitar o seguinte código:

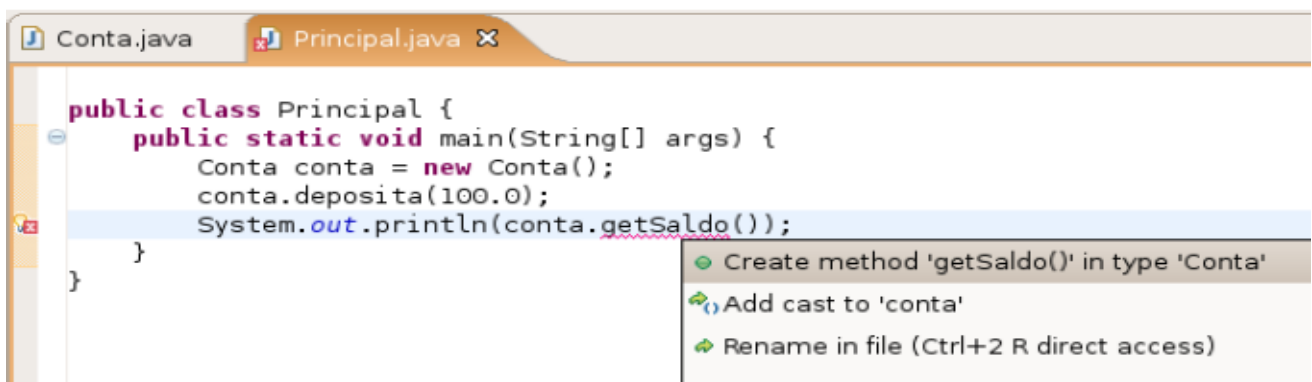
```
Conta conta = new Conta();
conta.deposita(100.0);
```

Observe que, na hora de invocar o método sobre o objeto conta, o Eclipse sugere os métodos possíveis. Este recurso é bastante útil, principalmente quando estivermos programando com classes que não são as nossas, como da API do Java. O Eclipse aciona este recurso quando você digita o ponto logo após um objeto (e você pode usar o Ctrl+Espaço para acioná-lo).

Vamos imprimir o saldo com System.out.println. Mas, mesmo nesse código, o Eclipse nos ajuda. Escreva syso e aperte Ctrl+Espaço que o Eclipse escreverá System.out.println() para você.

Para imprimir, chame o conta.getSaldo():
System.out.println(conta.getSaldo());

Note que o Eclipse acusará erro em getSaldo() porque este método não existe na classe Conta. Vamos usar Ctrl+1 em cima do erro para corrigir o problema:



O Eclipse sugere criar um método getSaldo() na classe Conta. Selecione esta opção e o método será inserido automaticamente.

```
public Object getSaldo() {
    // TODO Auto-generated method stub
    return null;
}
```

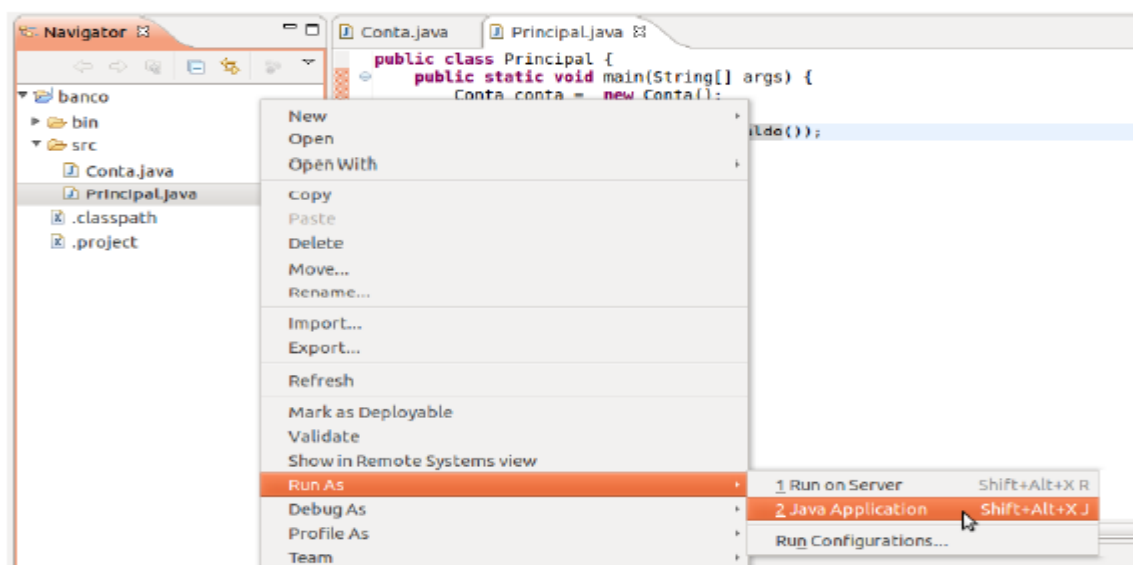
Ele gera um método não exatamente como queríamos, pois nem sempre há como o Eclipse ter de antemão informações suficientes para que ele acerta a assinatura do seu método. Modifique o método getSaldo como segue:

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

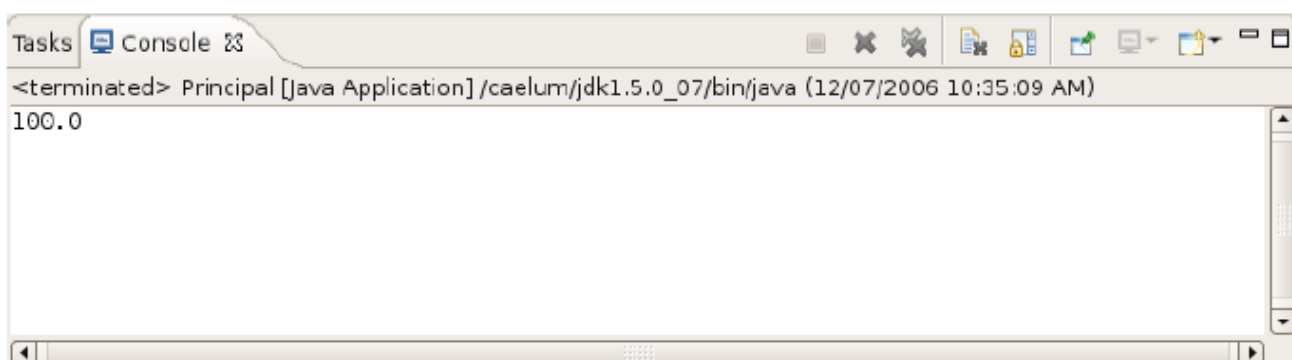
Esses pequenos recursos do Eclipse são de extrema utilidade. Dessa maneira, você pode programar sem se preocupar com métodos que ainda não existem, já que a qualquer momento ele pode gerar o esqueleto (a parte da assinatura do método).

8.6 – Executando o método MAIN

Vamos rodar o método main dessa nossa classe. No Eclipse, clique com o botão direito no arquivo Principal.java e vá em Run as... Java Application.



O Eclipse abrirá uma View chamada Console onde será apresentada a saída do seu programa:



Quando você precisar rodar de novo, basta clicar no ícone verde de play na toolbar, que roda o programa anterior. Ao lado desse ícone tem uma setinha onde são listados os 10 últimos executados.

8.7 – Pequenos Truques

O Eclipse possui muitos atalhos úteis para o programador. Sem dúvida os 3 mais importantes de conhecer e de praticar são:

- **Ctrl + 1** Aciona o quick fixes com sugestões para correção de erros.
- **Ctrl + Espaço** Completa códigos

- **Ctrl + 3** Aciona modo de descoberta de menu. Experimente digitar **Ctrl+3** e depois digitar **ggas** e **enter**. Ou então de **Ctrl + 3** e digite new class.

Existem dezenas de outros. Dentre os mais utilizados, escolhemos os seguintes para comentar:

- **Ctrl + F11** roda a última classe que você rodou. É o mesmo que clicar no ícone verde que parece um botão de play na barra de ferramentas.
- **Ctrl + PgUp** e **Ctrl + PgDown** Navega nas abas abertas. Útil quando estiver editando vários arquivos ao mesmo tempo.
- **Ctrl + Shift + F** Formata o código segundo as convenções do Java
- **Ctrl + M** Expande a View atual para a tela toda (mesmo efeito de dar dois cliques no título da View)
- **Ctrl + Shift + L** Exibe todos os atalhos possíveis.
- **Ctrl + O** Exibe um outline para rápida navegação
- **Alt + Shift + X e depois J** Roda o main da classe atual. Péssimo para pressionar! Mais fácil você digitar **Control+3** e depois digitar Run!. Abuse desde já do **Control+3**

8.8 – Exercícios no Eclipse

1) Dentro do projeto banco, crie as classes ContaCorrente e ContaPoupanca no nosso projeto do Eclipse. Na classe Conta, crie os métodos atualiza e saca como no capítulo anterior. Desta vez, tente abusar do control + espaço e control + 3.

Por exemplo:

```
publ<ctrl espaço> v<ctrl espaço> atualiza(do<ctrl espaço> taxa){
```

Repare que até mesmo nomes de variáveis, ele cria para você! Acompanhe as dicas do instrutor. Muitas vezes, ao criarmos um objeto, nem mesmo declaramos a variável:

```
new ContaCorrente();
```

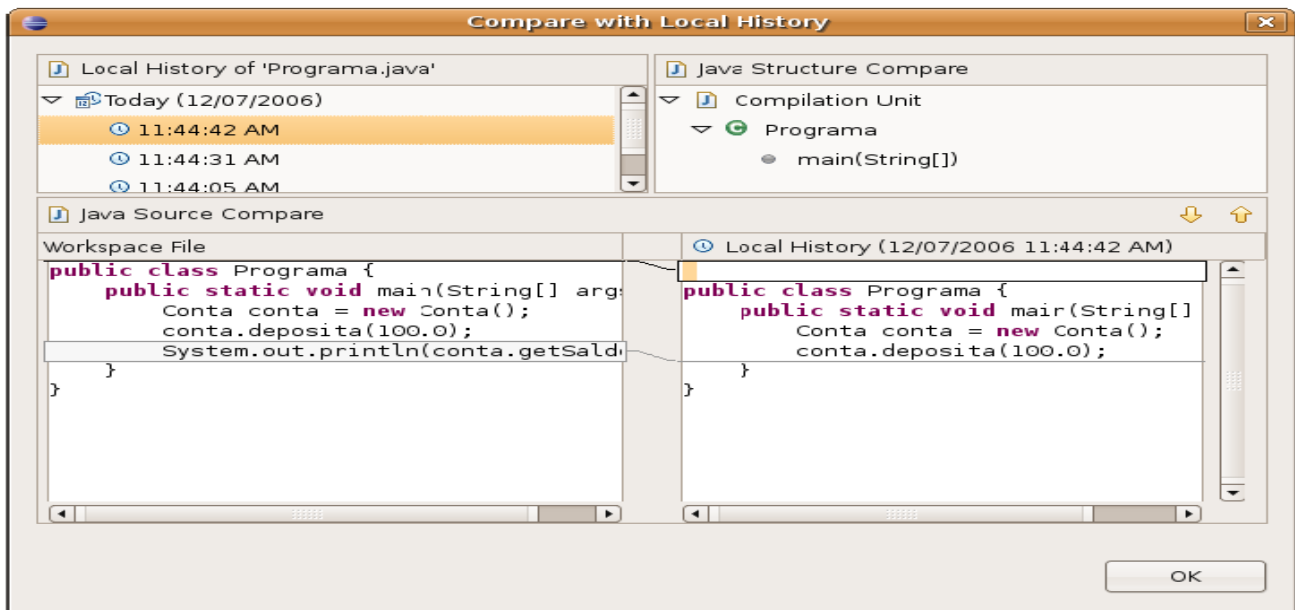
Vá nessa linha e dê control + 3. Ele vai sugerir e declarará a variável pra você.

2) Imagine que queremos criar um setter do saldo para a classe Conta. Dentro da classe Conta, digite: setSa<ctrl + espaço>

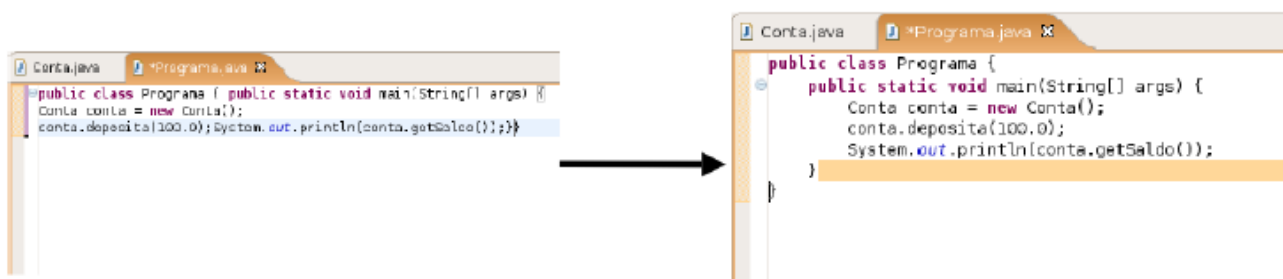
O mesmo vale no caso de você querer reescrever um método. Dentro de ContaCorrente faça: atua<ControlEspaco>

3) Vá na sua classe que tem o main e segure o CONTROL apertado enquanto você passa o mouse sobre o seu código. Repare que tudo virou hyperlink. Clique em um método que você está invocando na classe Conta. Você pode conseguir o mesmo efeito, de abrir o arquivo no qual o método foi declarado, de uma maneira ainda mais prática: sem usar o mouse, quando o cursor estiver sobre o que você quer Analisar, simplesmente clique F3.

4) Dê um clique da direita em um arquivo no navigator. Escolha **CompareWith -> Local History**. O que é esta tela?



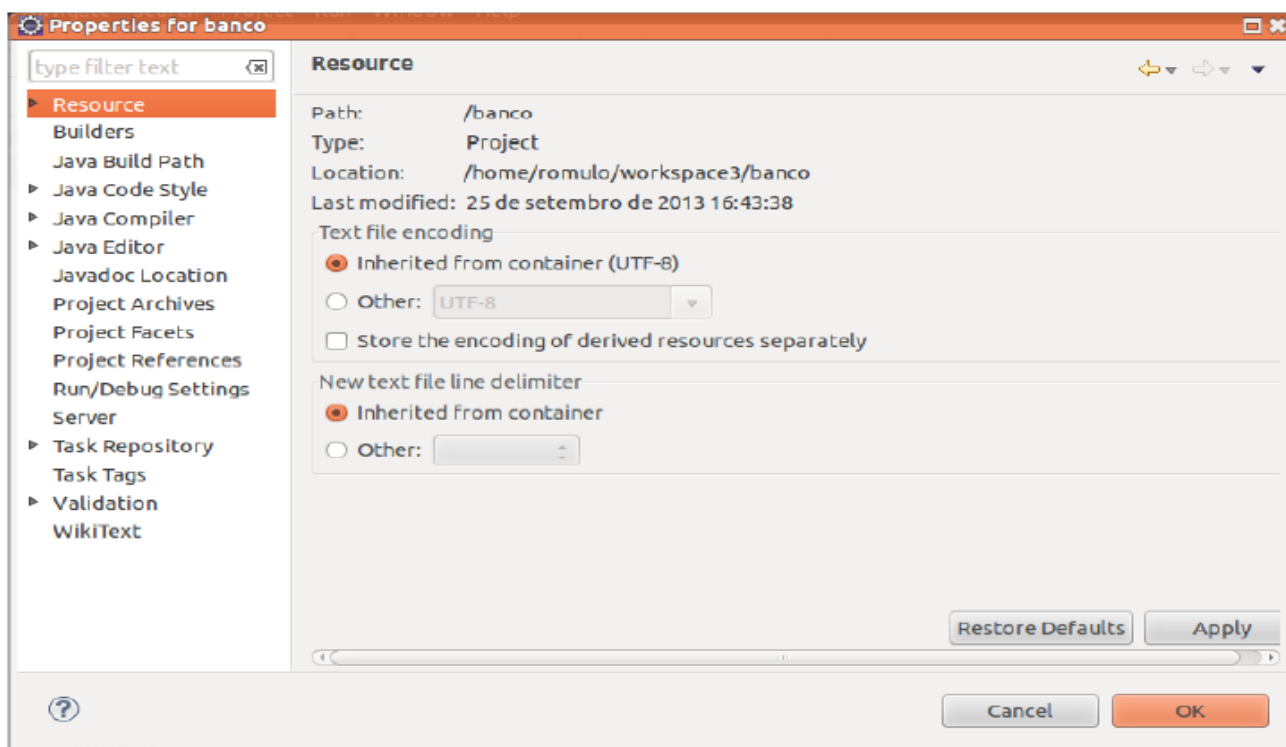
5) Use o Control + Shift + F para formatar o seu código. Dessa maneira, ele vai arrumar a bagunça de espaçamento e enters do seu código.



6) (opcional) Crie no seu projeto a classe AtualizadorDeContas como no capítulo anterior.

7) (opcional) O que são os arquivos .project e .classpath? Leia o conteúdo deles.

8) (opcional) Clique da direita no projeto, propriedades. É uma das telas mais importantes do Eclipse, onde você pode configurar diversas informações para o seu projeto, como compilador, versões, formatador e outros.



9 – Classes Abstratas

9.1 – 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 facilidades 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 o nosso ControleDeBonificacao:

```
class ControleDeBonificacoes {
    private double totalDeBonificacoes = 0;
    public void registra(Funcionario f) {
        System.out.println("Adicionando bonificação do funcionario: " + f);
        this.totalDeBonificacoes += f.getBonificacao();
    }
    public double getTotalDeBonificacoes() {
```

```

        return this.totalDeBonificacoes;
    }
}

```

Nosso método registra recebe qualquer referência do tipo Funcionario, isto é, podem 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 registra para receber cada um dos tipos de Funcionario, um para Gerente, um para Diretor, etc. Repare que perder esse poder é muito pior do que a pequena vantagem que a herança traz em herdar código.

Porém, em alguns sistemas, como é o nosso caso, usamos uma classe com apenas esses intuitos: de economizar um pouco código e ganhar polimorfismo para criar métodos mais genéricos, 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: nesse caso, faz sim e é muito útil. Referenciando Funcionario temos o polimorfismo de referência, já que podemos receber qualquer objeto que seja um Funcionario. Porém, dar new em Funcionario pode não fazer sentido, isto é, não queremos receber um objeto do tipo Funcionario, mas sim 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?

```

Vejam os outros casos 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.

9.2 – Classe Abstrata

O que, exatamente, vem a ser a nossa classe Funcionario? Nossa empresa tem apenas Diretores, Gerentes, Secretárias, etc. Ela é uma classe que apenas idealiza um tipo, define apenas um rascunho. Para o nosso sistema, é inadmissível que um objeto seja 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).

Usamos a palavra chave abstract para impedir que ela possa ser instanciada. Esse é o efeito direto 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;
    }
}

```

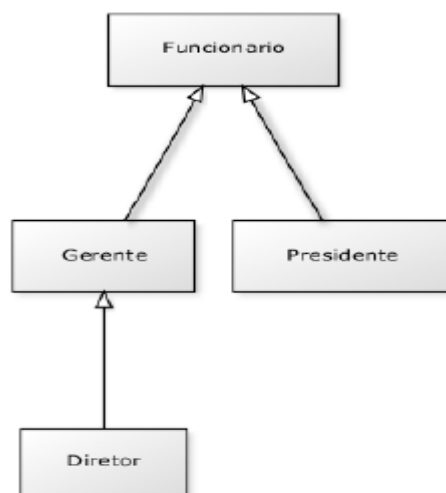
```
// outros atributos e métodos comuns a todos Funcionarios
}
```

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. Se ela não pode ser instanciada, para que serve? Serve para o polimorfismo e herança dos atributos e métodos, que são recursos muito poderosos, como já vimos. Vamos então herdar dessa classe, reescrevendo o método `getBonificacao`:

```
class Gerente extends Funcionario {
    public double 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. Fique claro que a nossa decisão de transformar `Funcionario` em uma classe abstrata dependeu do nosso domínio. Pode ser que, em um sistema com classes similares, faça sentido que uma classe análoga a `Funcionario` seja concreta.

9.3 – Métodos Abstratos

Se o método `getBonificacao` não fosse reescrito, ele seria herdado da classe mãe, fazendo com que devolvesse o salário mais 20%.

Levando em consideração que cada funcionário em nosso sistema tem uma regra totalmente diferente para ser bonificado, faz algum sentido ter esse método na classe `Funcionario`? Será que existe uma bonificação padrão para todo tipo de `Funcionario`? Parece que 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. Será que então devemos retornar um código, como um número negativo? Isso não resolve o problema: se esquecermos de reescrever esse método, teremos dados errados sendo utilizados como bônus.

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, isto é, que não forem abstratas) devem reescrever esse método ou não compilarão. É como se você herdasse a responsabilidade de ter aquele método. As 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. Por que não colocar corpo algum? Porque esse método nunca vai ser chamado, sempre que 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:

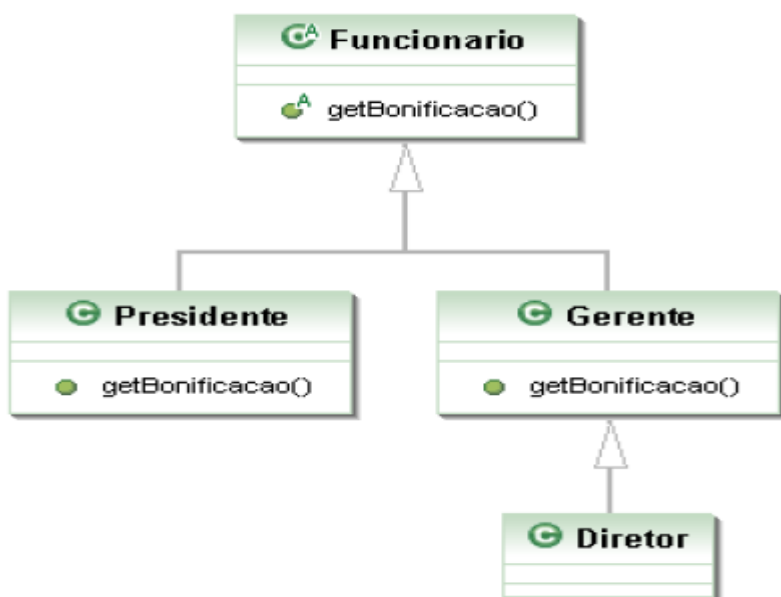
```
public void registra(Funcionario f) {
    System.out.println("Adicionando bonificação do funcionario: " + f);
    this.totalDeBonificacoes += f.getBonificacao();
}
```

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.

9.4 – Aumentando o exemplo

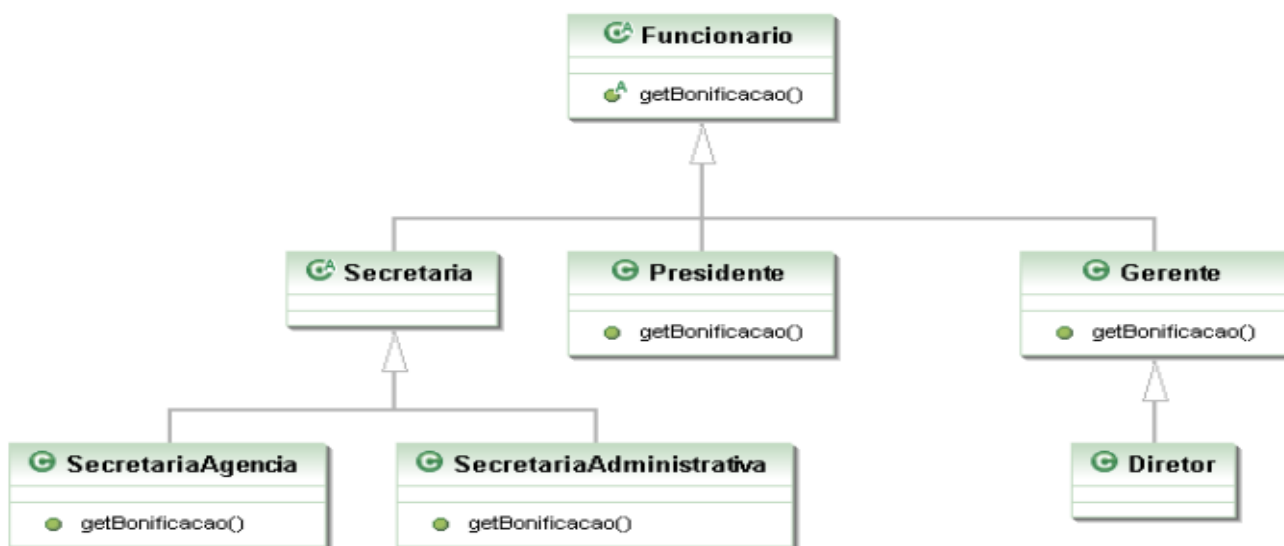
E se, no nosso exemplo de empresa, tivéssemos o seguinte diagrama de classes com os seguintes métodos:



Ou seja, tenho a classe abstrata `Funcionario`, com o método abstrato `getBonificacao()`; as classes `Gerente` e `Presidente` estendendo `Funcionario` e implementando o método `getBonificacao()`; e, por fim, a classe `Diretor`, que estende `Gerente`, mas não implementa o método `getBonificacao()`.

Essas classes vão compilar? Vão rodar?

A resposta é sim. E, além de tudo, farão exatamente o que nós queremos, pois, quando `Gerente` e `Presidente` possuem os métodos perfeitamente implementados, a classe `Diretor`, que não possui o método implementado, vai usar a implementação herdada de `Gerente`.



E esse diagrama, no qual incluímos uma classe abstrata `Secretaria` sem o método `getBonificacao()`, que é estendida por mais duas classes (`SecretariaAdministrativa`, `SecretariaAgencia`) que, por sua vez, implementam o método `getBonificacao()`, vai compilar? Vai rodar?

De novo, a resposta é sim, pois `Secretaria` é uma classe abstrata e, por isso, o Java tem certeza de que ninguém vai conseguir instanciá-la e, muito menos, chamar o método `getBonificacao()` dela.

Lembrando que, nesse caso, não precisamos nem ao menos escrever o método abstrato `getBonificacao` na classe `Secretaria`.

Se eu não reescrever um método abstrato da minha classe mãe, o código não compilará. Mas posso, em vez disso, declarar a classe como abstrata!

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 bastantes classes abstratas, sendo um exemplo real de uso desse recurso, que vai melhorar o entendimento delas. (classe `InputStream` e suas filhas).

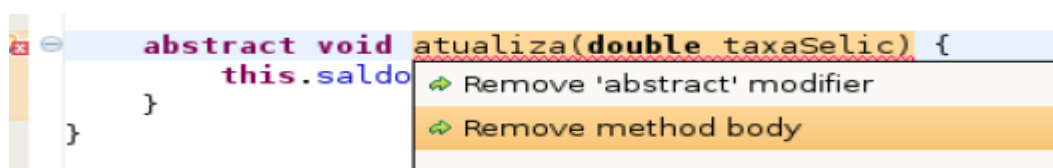
9.5 – Exercícios: Classes Abstratas

1) Repare que a nossa classe `Conta` é uma excelente candidata para uma classe abstrata. Por quê? Que métodos seriam interessantes candidatos a serem abstratos? Transforme a classe `Conta` em abstrata, repare o que acontece no seu main já existente do `TestaContas`.

```
public abstract class Conta {
    // ...
}
```

2) Para que o código do main volte a compilar, troque o `new Conta()` por `new ContaCorrente()`. Se 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) Apenas para entender melhor o `abstract`, comente o método `atualiza()` da `ContaPoupanca`, dessa forma ele herdará o método diretamente de `Conta`. Transforme o método `atualiza()` da classe `Conta` em abstrato. Repare que, ao colocar a palavra chave `abstract` ao lado do método, o Eclipse rapidamente vai sugerir que você deve remover o corpo (body) do método com um quick fix:

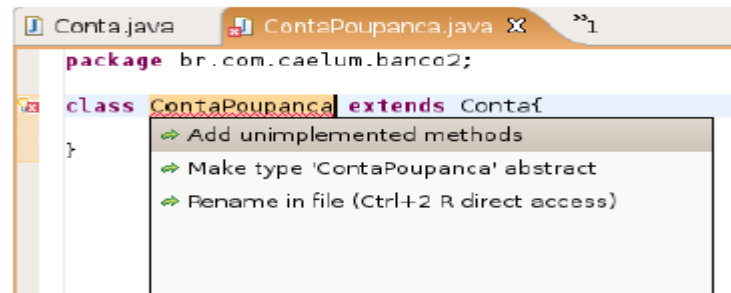


Sua classe `Conta` deve ficar parecida com:

```
public abstract class Conta {
    // atributos e métodos que já existiam
    public abstract void atualiza(double taxaSelic);
}
```

Qual é o problema com a classe `ContaPoupanca`?

4) Reescreva o método `atualiza()` na classe `ContaPoupanca` para que a classe possa compilar normalmente. O eclipse também sugere isso como um quick fix:



5) (opcional) Existe outra maneira de a 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 apagarmos esse método da classe Conta e deixarmos o método atualiza nas filhas?

7) (opcional) Posso chamar um método abstrato de dentro de um outro método da própria classe abstrata? Um exemplo: o mostra do Funcionario pode invocar this.getBonificacao()?

10 - Interfaces

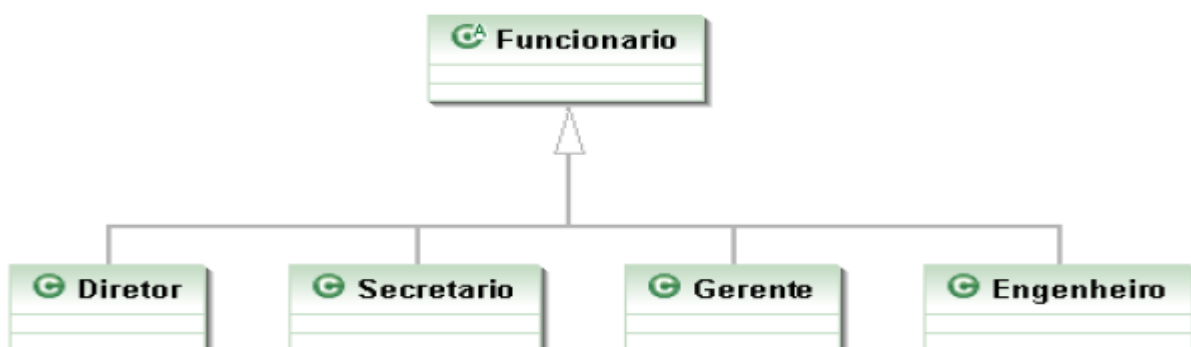
10.1 – Aumentando nosso exemplo

Imagine que um Sistema de Controle do Banco pode ser acessado, além de pelos 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 também se o departamento 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 colocá-lo dentro do sistema.

```
class SistemaInterno {
    void login(Funcionario funcionario) {
        // invocar o método autentica?
        // não da! Nem todo Funcionario tem
    }
}
```

O SistemaInterno aceita qualquer tipo de Funcionario, tendo ele acesso ao sistema ou não, mas note que nem todo Funcionario possui o método autentica. Isso nos impede de chamar esse método com uma referência apenas a Funcionario (haveria um erro de compilação). O que fazer então?

```
class SistemaInterno {
    void login(Funcionario funcionario) {
        funcionario.autentica(...); // não compila
    }
}
```

Uma possibilidade é criar dois métodos login no SistemaInterno: um para receber Diretor e outro para receber Gerente. Já vimos que essa não é uma boa escolha. Por quê?

```
class SistemaInterno {
    // design problemático
    void login(Diretor funcionario) {
        funcionario.autentica(...);
    }
    // design problemático
    void login(Gerente funcionario) {
        funcionario.autentica(...);
    }
}
```

Cada vez que criarmos uma nova classe de Funcionario que é autenticável, precisaríamos adicionar um novo método de login no SistemaInterno. Em Java, métodos podem ter o mesmo nome desde que não sejam ambíguos, isto 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). Uma solução mais interessante seria criar uma classe no meio da árvore de herança,

FuncionarioAutenticavel:

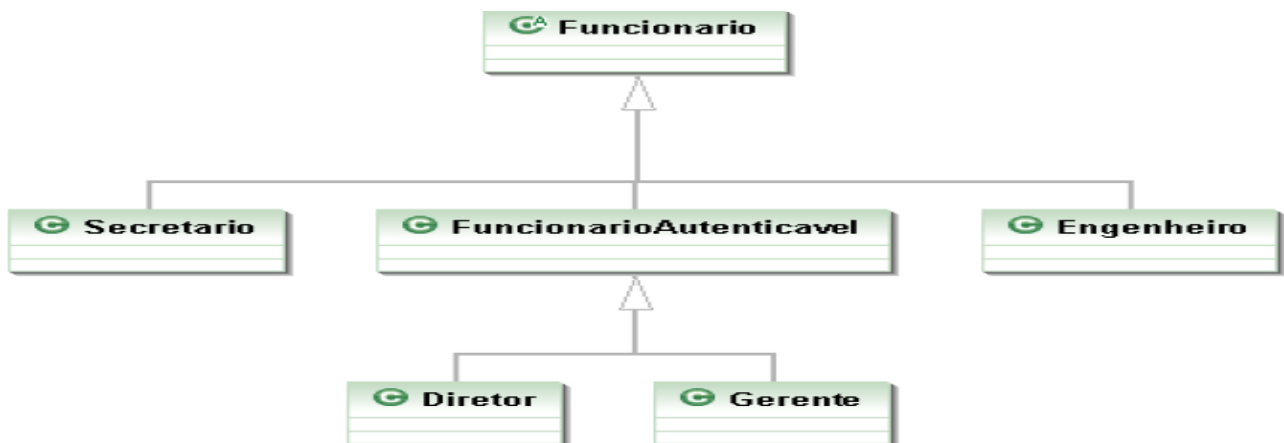
```
class FuncionarioAutenticavel extends Funcionario {
    public boolean autentica(int senha) {
        // faz autenticacao padrão
    }
    // outros atributos e métodos
}
```

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

```

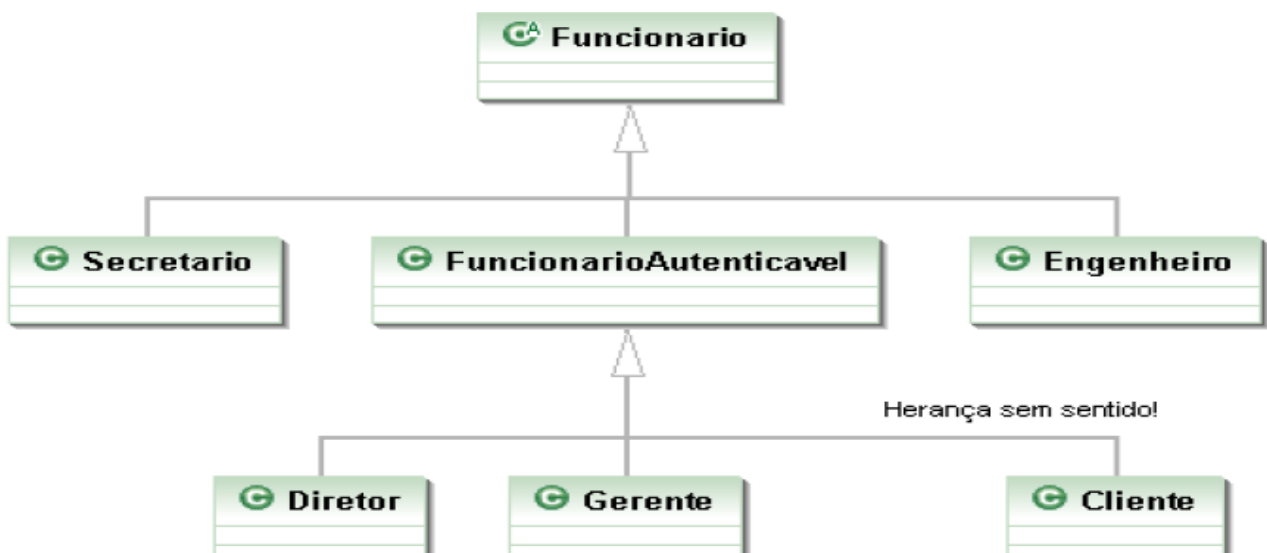
class SistemaInterno {
    void login(FuncionarioAutenticavel fa) {
        int senha = //pega senha de um lugar, ou de um scanner de polegar
        // aqui eu posso chamar o autentica!
        // Pois todo FuncionarioAutenticavel tem
        boolean ok = fa.autentica(senha);
    }
}

```



Repare que **FuncionarioAutenticavel** é uma forte candidata a classe abstrata. Mais ainda, o método **autentica** poderia ser um método abstrato. O uso de herança resolve esse caso, mas vamos a uma outra situação um pouco mais complexa: precisamos que todos os clientes também tenham acesso ao **SistemaInterno**. O que fazer? Uma opção é criar outro método **login** em **SistemaInterno**: mas já descartamos essa anteriormente.

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 trará diversos outros. **Cliente** definitivamente **não é** **FuncionarioAutenticavel**. Se você fizer isso, o **Cliente** terá, por exemplo, um método **getBonificacao**, um atributo **salario** e outros membros que não fazem o menor sentido para esta classe! Não faça herança quando a relação não é estritamente “é um”.



Como resolver essa situação? Note que conhecer a sintaxe da linguagem não é o suficiente, precisamos estruturar/desenhar bem a nossa estrutura de classes.

10.2 – 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 na qual essas classes garantissem a existência de um determinado método, através de um contrato, resolveríamos o problema. Toda classe define 2 itens:

- o que uma classe faz (as assinaturas dos métodos)
- como uma classe faz essas tarefas (o corpo dos métodos e atributos privados)

Podemos criar um “contrato” que define tudo o que uma classe deve fazer se quiser ter um determinado status. Imagine:

contrato Autenticavel:

quem quiser ser Autenticavel precisa saber fazer:

1.autenticar dada uma senha, devolvendo um booleano

Quem quiser, pode “assinar” esse contrato, sendo assim obrigado a explicar como será feita essa autenticação. A vantagem é que, se um Gerente assinar esse contrato, podemos nos referenciar a um Gerente como um Autenticavel. Podemos criar esse contrato em Java!

```
interface Autenticavel {  
    boolean autentica(int senha);  
}
```

Chama-se interface pois é a maneira pela qual poderemos conversar com um Autenticavel. Interface é a maneira através da qual conversamos com um objeto. Lemos a interface da seguinte maneira: "quem desejar ser autenticável precisa saber autenticar dado um inteiro e retornando um booleano". Ela é um contrato onde quem assina se responsabiliza por implementar esses métodos (cumprir o contrato).

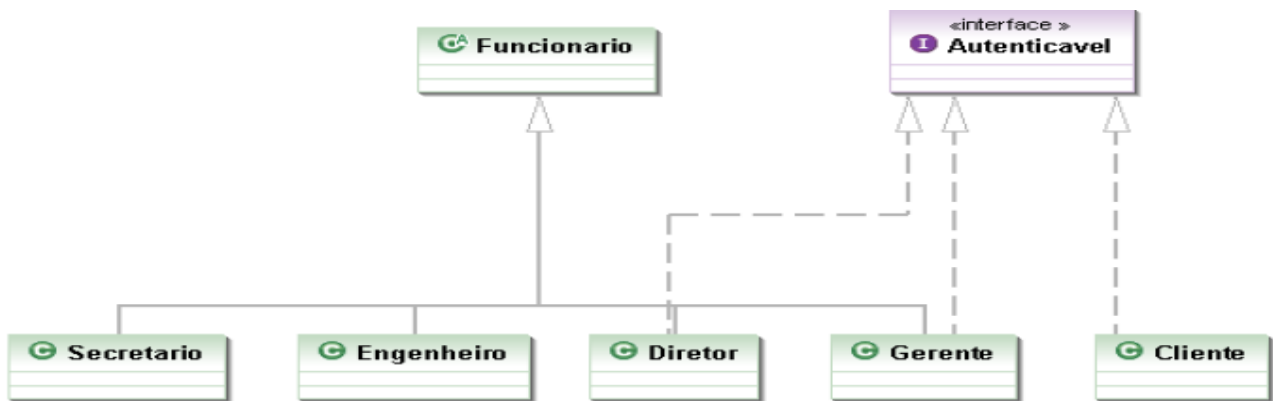
Uma interface pode definir uma série de métodos, mas nunca conter implementação deles. Ela só expõe **o que o objeto deve fazer**, e não **como ele faz**, nem **o que ele tem**. **Como ele faz** vai ser definido em uma **implementação** dessa interface.

E o Gerente pode “assinar” o contrato, ou seja, **implementar** a interface. No momento em que ele implementa essa interface, ele precisa escrever os métodos pedidos pela interface (muito parecido com o 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 possíveis verificações, como saber se esse
        // departamento do gerente tem acesso ao Sistema
        return true;
    }
}

```



O implements pode ser lido da seguinte maneira: “A classe Gerente se compromete a ser tratada como Autenticavel, sendo obrigada a ter os métodos necessários, definidos neste contrato”.

A partir de agora, podemos tratar um Gerente como sendo um Autenticavel. Ganhamos mais 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 implemente Autenticavel, direta ou indiretamente:

```

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

```

Novamente, a utilização mais comum seria receber por argumento, como no nosso SistemaInterno:

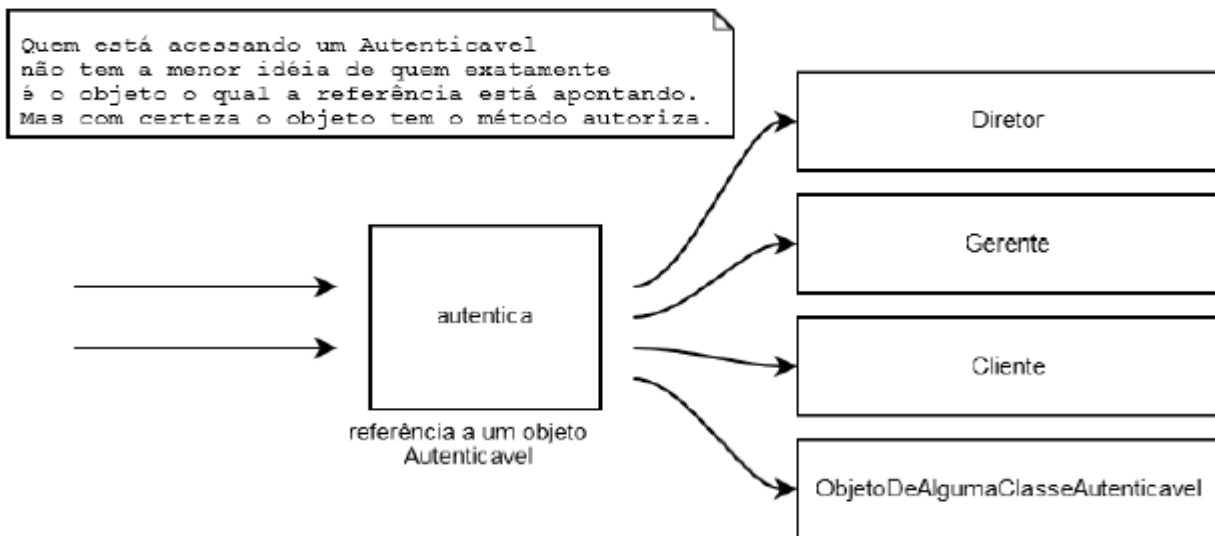
```

class SistemaInterno {
    void login(Autenticavel a) {
        int senha = // pega senha de um lugar, ou de um scanner de polegar
        boolean ok = a.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 {
    // métodos e atributos, além de obrigatoriamente ter o autentica
}
```



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.

Qualquer Autenticavel passado para o SistemaInterno está bom para nós. Repare que pouco importa quem o objeto referenciado realmente é, pois ele tem um método autentica que é o necessário para nosso SistemaInterno funcionar corretamente. Aliás, qualquer outra classe que futuramente implemente essa interface poderá ser passada como argumento aqui.

```
Autenticavel diretor = new Diretor();
Autenticavel gerente = new Gerente();
```

Ou, se 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.

```
class SistemaInterno {
    void login(Autenticavel a) {
        // não importa se ele é um gerente ou diretor
        // será que é um fornecedor?
        // Eu, o programador do SistemaInterno, não me preocupo
        // Invocarei o método autentica
    }
}
```

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. Lembre-se: a interface define que todos vão saber se autenticar (o que ele faz), enquanto 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 do que **como ele faz**. Aqueles que seguem essa regra, terão sistemas mais fáceis de manter e modificar. Como você já

percebeu, esta é uma das idéias principais que queremos passar e, provavelmente, a mais importante de todo esse curso.

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

10.3 – Dificuldade no aprendizado das interfaces

Interfaces representam uma barreira no aprendizado do Java: parece que estamos escrevendo um código que não serve pra nada, já que teremos essa linha (a assinatura do método) 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. **Não é apenas um código de prototipação, um cabeçalho!**

Os mais radicais 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 deste material acham tal medida radical demais, porém o uso de interfaces em vez de herança é amplamente aconselhado. Você pode encontrar mais informações sobre o assunto nos livros Design Patterns, Refactoring e Effective Java.

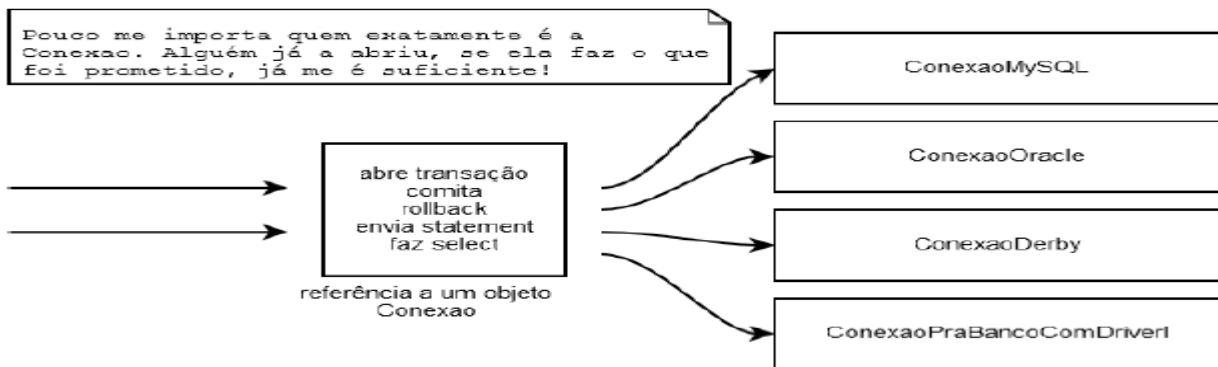
No livro Design Patterns, logo no início, os autores citam ó regras “de ouro”. Uma é “evite herança, prefira composição” e a outra, “ programe voltado a interface e não à implementação”.

Veremos o uso de interfaces no capítulo de coleções, o que melhora o entendimento do assunto. O exemplo da interface Comparable também é muito esclarecedor, onde enxergamos o reaproveitamento de código através das interfaces, além do encapsulamento. Para o método Collections.sort(), pouco importa quem vai ser passado como argumento. Para ele, basta que a coleção seja de objetos comparáveis. Ele pode ordenar Elefante, Conexao ou ContaCorrente, desde que implementem Comparable.

10.4 – Exemplo Interessante: Conexões com Banco de Dados

Como fazer com que todas as chamadas para bancos de dados diferentes respeitem a mesma regra? Usando interfaces! Imagine uma interface Conexao contendo todos os métodos necessários para a comunicação e troca de dados com um banco de dados. Cada banco de dados fica encarregado de criar a sua implementação para essa interface.

Quem for usar uma Conexao não precisa se importar com qual objeto exatamente está trabalhando, já que ele vai cumprir o papel que toda Conexao deve ter. Não importa se é uma conexão com um Oracle ou MySQL.



Apesar do `java.sql.Connection` não trabalhar bem assim, a ideia é muito similar, porém as conexões vêm de uma factory chamada `DriverManager`.

10.5 – Exercícios: Interfaces

1) A sintaxe do uso de interfaces pode parecer muito estranha, à primeira vista.

Vamos começar com um exercício para praticar a sintaxe. Crie um projeto interfaces e crie a interface `AreaCalculavel`:

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

Queremos criar algumas classes que são `AreaCalculavel`:

```
class Quadrado implements AreaCalculavel {
    private int lado;
    public Quadrado(int lado) {
        this.lado = lado;
    }
    public double calculaArea() {
        return this.lado * this.lado;
    }
}
```

```
class Retangulo implements AreaCalculavel {
    private int largura;
    private int altura;
    public Retangulo(int largura, int altura) {
        this.largura = largura;
        this.altura = altura;
    }
    public double calculaArea() {
        return this.largura * this.altura;
    }
}
```

Repare que, aqui, se você tivesse usado herança, não iria ganhar muito, já que cada implementação é totalmente diferente da outra: um `Quadrado`, um `Retangulo` e um `Circulo` têm atributos e métodos **bem** diferentes.

Mas, mesmo que eles tivessem atributos em comum, utilizar interfaces é uma maneira muito mais elegante de modelar suas classes. Elas também trazem vantagens em não acoplar as classes. Uma vez que herança através de classes traz muito acoplamento, muitos autores renomados dizem que, na maioria dos casos, **herança quebra o encapsulamento**.

Crie a seguinte classe de Teste. Repare no polimorfismo. Poderíamos passar esses objetos como argumento para alguém que aceitasse `AreaCalculavel` como argumento:

```
class Teste {
    public static void main(String[] args) {
        AreaCalculavel a = new Retangulo(3,2);
        System.out.println(a.calculaArea());
    }
}
```

Opcionalmente, crie a classe `Circulo`:

```
class Circulo implements AreaCalculavel {
    // ... atributos (raio) e métodos (calculaArea)
}
```

Utilize `Math.PI * raio * raio` para calcular a área.

2) Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso, vamos criar uma interface no nosso projeto banco já existente:

```
interface Tributavel {
    double calculaTributos();
}
```

Lemos essa interface da seguinte maneira: “todos que quiserem ser tributável precisam saber calcular tributos, devolvendo um `double`”. Alguns bens são tributáveis e outros não, `ContaPoupanca` não é tributável, já para `ContaCorrente` você precisa pagar 1% da conta e o `SeguroDeVida` tem uma taxa fixa de 42 reais.

Aproveite o Eclipse! Quando você escrever `implements Tributavel` na classe `ContaCorrente`, o quick fix do Eclipse vai sugerir que você reescreva o método; escolha essa opção e, depois, preencha o corpo do método adequadamente:

```
class ContaCorrente extends Conta implements Tributavel {
    // outros atributos e métodos
    public double calculaTributos() {
        return this.getSaldo() * 0.01;
    }
}
```

Crie a classe `SeguroDeVida`, aproveitando novamente do Eclipse, para obter:

```
class SeguroDeVida implements Tributavel {
    public double calculaTributos() {
        return 42;
    }
}
```

Vamos criar uma classe TestaTributavel com um método main para testar o nosso exemplo:

```
class TestaTributavel {
    public static void main(String[] args) {
        ContaCorrente cc = new ContaCorrente();
        cc.deposita(100);
        System.out.println(cc.calculaTributos());

        // testando polimorfismo:
        Tributavel t = cc;
        System.out.println(t.calculaTributos());
    }
}
```

Tente chamar o método getSaldo através da referência t, o que ocorre? Por quê?

A linha em que atribuímos cc a um Tributavel é apenas para você enxergar que é possível fazê-lo.

Nesse nosso caso, isso não tem uma utilidade. Essa possibilidade será útil para o próximo exercício.

11 – Exceções e Controle de erros

11.1 – Motivação

Voltando às Contas que criamos no capítulo 6, o que aconteceria ao tentar chamar o método saca com um valor fora do limite? O sistema mostraria uma mensagem de erro, mas quem chamou o método saca não saberá que isso aconteceu. Como avisar aquele que chamou o método de 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 o que deveria, ele precisa, ao menos, avisar ao usuário que o saque 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 do que a classe sinalizar que um erro ocorreu. Asolução mais simples utilizada antigamente é a demarcar o retorno de um método como boolean e retornar true, se tudo ocorreu da maneira planejada, ou false, caso contrário:

```
boolean saca(double quantidade) {
    // posso sacar até saldo+limite
    if (quantidade > this.saldo + this.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.deposita(100);
minhaConta.setLimite(100);
if (!minhaConta.saca(1000)) {
    System.out.println("Não saquei");
}

```

Repare que tivemos de lembrar de testar o retorno do método, mas não somos obrigados a fazer isso. Esquecer de testar o retorno desse método teria consequências drásticas: a máquina de auto-atendimento poderia vir a liberar a quantia desejada de dinheiro, mesmo que o sistema não tivesse conseguido efetuar o método saca com sucesso, como no exemplo a seguir:

```

Conta minhaConta = new Conta();
minhaConta.deposita(100);
// ...
double valor = 5000;
minhaConta.saca(valor); // vai retornar false, mas ninguém verifica!
caixaEletronico.emite(valor);

```

Mesmo invocando o método e tratando o retorno de maneira correta, o que faríamos se fosse necessário sinalizar quando o usuário passou um valor negativo como **quantidade**? Uma solução seria alterar o retorno de boolean para int e retornar o código do erro que ocorreu. Isso é considerado uma má prática (conhecida também como uso de “magic numbers”).

Além de você perder o retorno do método, o valor devolvido é “mágico” e só legível perante extensa documentação, além de não obrigar o programador a tratar esse retorno e, no caso de esquecer isso, seu programa continuará rodando já num estado inconsistente. Repare o que aconteceria 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 Cliente();
        cliente.setId(id);
        // cliente.setNome("nome do cliente");
        return cliente;
    }
}

```

Por esses 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** à regra. Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho

ou inesperado no sistema.

11.2 – Exercício para começar com os conceitos

Antes de resolvermos o nosso problema, vamos ver como a Java Virtual Machine age ao se deparar com situações inesperadas, como divisão por zero ou acesso a um índice da array que não existe.

1) Para aprendermos os conceitos básicos das exceptions do Java, teste o seguinte código você mesmo:

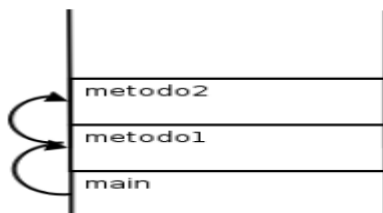
```
class TesteErro {
    public static void main(String[] args) {
        System.out.println("inicio do main");
        metodo1();
        System.out.println("fim do main");
    }

    static void metodo1() {
        System.out.println("inicio do metodo1");
        metodo2();
        System.out.println("fim do metodo1");
    }

    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");
    }
}
```

Repare o método main chamando metodo1 e esse, por sua vez, chamando o metodo2. Cada um desses métodos pode ter suas próprias variáveis locais, isto é: o metodo1 não enxerga as variáveis declaradas dentro do main e por aí em diante.

Como o Java (e muitas das outras linguagens) faz isso? Toda invocação de método é empilhada em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da **pilha de execução** (stack): basta remover o marcador que está no topo da pilha:



Porém, o nosso metodo2 propositadamente possui um enorme problema: está acessando um índice de array indevido para esse caso; o índice estará fora dos limites da array quando chegar em 10! Rode o código. Qual é a saída? O que isso representa? O que ela indica?



```

Console X
<terminated> Teste [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:44:42 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
Exception in thread 'main' java.lang.ArrayIndexOutOfBoundsException: 10
    at Teste.metodo2(Teste.java:18)
    at Teste.metodo1(Teste.java:10)
    at Teste.main(Teste.java:4)
  
```

Essa é o conhecido **rastro da pilha** (stacktrace). É uma saída importantíssima para o programador – tanto que, em qualquer fórum ou lista de discussão, é comum os programadores enviarem, juntamente com a descrição do problema, essa stacktrace. Mas por que isso aconteceu? O sistema de exceções do Java funciona da seguinte maneira: quando uma exceção é **lançada** (throw), a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao **tentar** executar esse trecho de código. Como podemos ver, o metodo2 não toma nenhuma medida diferente do que vimos até agora.

Como o metodo2 não está **tratando** esse problema, a JVM pára a execução dele anormalmente, sem esperar ele terminar, e volta um stackframe pra baixo, onde será feita nova verificação: “o metodo1 está se precavendo de um problema chamado ArrayIndexOutOfBoundsException?” “Não...” Volta para o main, onde também não há proteção, então a JVM morre (na verdade, quem morre é apenas a Thread corrente, veremos mais para frente).

Obviamente, aqui estamos forçando esse caso e não faria sentido tomarmos cuidado com ele. É fácil arrumar um problema desses: basta percorrermos a array no máximo até o seu length. Porém, apenas para entender o controle de fluxo de uma Exception, vamos colocar o código que vai **tentar** (try) executar o bloco perigoso e, caso o problema seja do tipo ArrayIndexOutOfBoundsException, ele será **pego** (caught). Repare que é interessante que cada exceção no Java tenha um tipo... ela pode ter atributos e métodos.

2) Adicione um try/catch em volta do for, pegando ArrayIndexOutOfBoundsException. O que o código imprime?

```

try {
    for (int i = 0; i <= 15; i++) {
        array[i] = i;
        System.out.println(i);
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
  
```



```
}
```



```

Console X
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4:50:20 PM)
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
fim do metodo2
fim do metodo1
fim do main

```

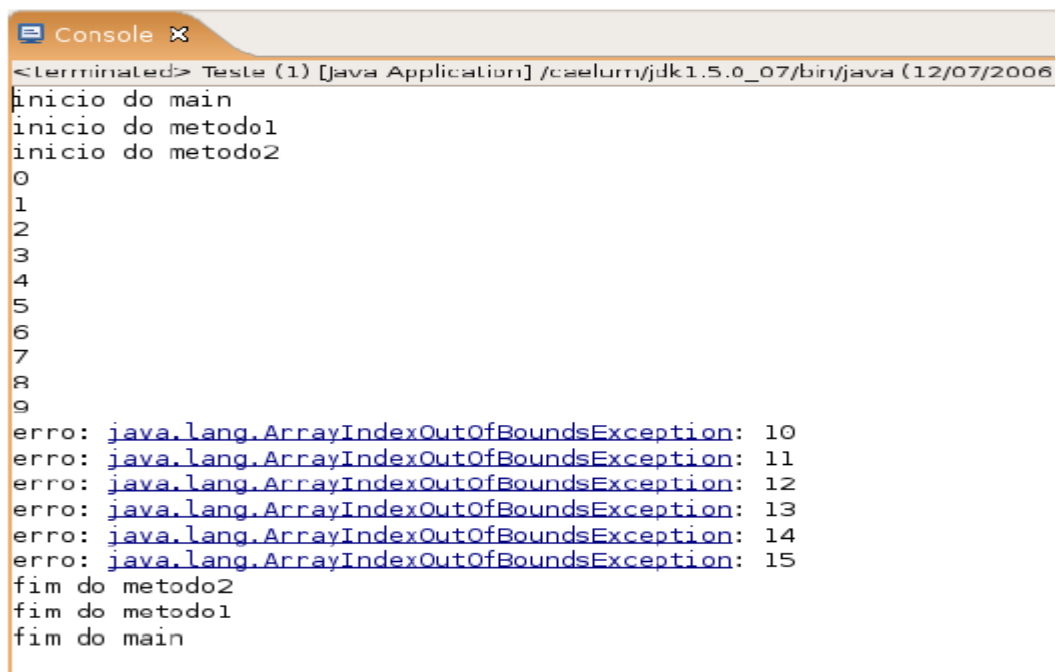
3) Em vez de fazer o try em torno do for inteiro, tente apenas com o bloco de dentro do for:

```

for (int i = 0; i <= 15; i++) {
    try {
        array[i] = i;
        System.out.println(i);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("erro: " + e);
    }
}

```

Qual é a diferença?




```

Console X
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
erro: java.lang.ArrayIndexOutOfBoundsException: 11
erro: java.lang.ArrayIndexOutOfBoundsException: 12
erro: java.lang.ArrayIndexOutOfBoundsException: 13
erro: java.lang.ArrayIndexOutOfBoundsException: 14
erro: java.lang.ArrayIndexOutOfBoundsException: 15
fim do metodo2
fim do metodo1
fim do main

```

4) Retire o try/catch e coloque ele em volta da chamada do metodo2.

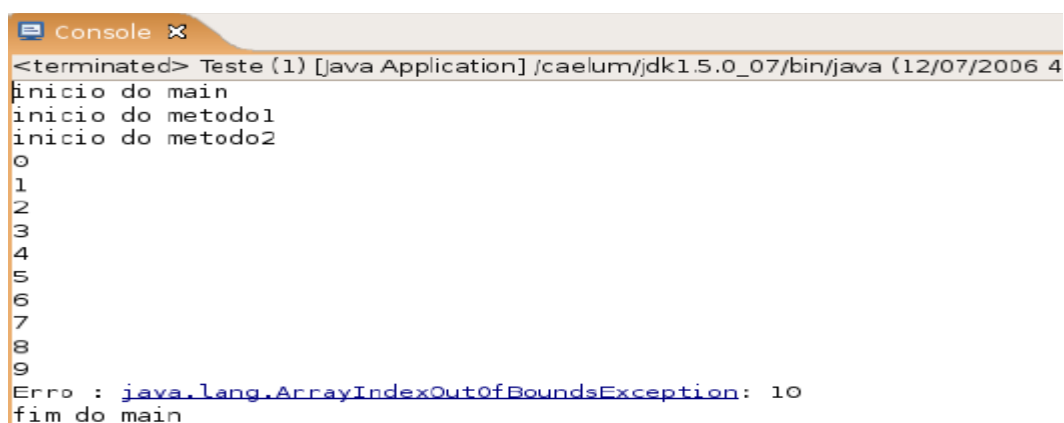
```
System.out.println("inicio do metodo1");
try {
    metodo2();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}
System.out.println("fim do metodo1");
```



```
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
erro: java.lang.ArrayIndexOutOfBoundsException: 10
fim do metodo1
fim do main
```

5) Faça o mesmo, retirando o try/catch novamente e colocando em volta da chamada do metodo1. Rode os códigos, o que acontece?

```
System.out.println("inicio do main");
try {
    metodo1();
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Erro : "+e);
}
System.out.println("fim do main");
```



```
<terminated> Teste (1) [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 4
inicio do main
inicio do metodo1
inicio do metodo2
0
1
2
3
4
5
6
7
8
9
Erro : java.lang.ArrayIndexOutOfBoundsException: 10
fim do main
```

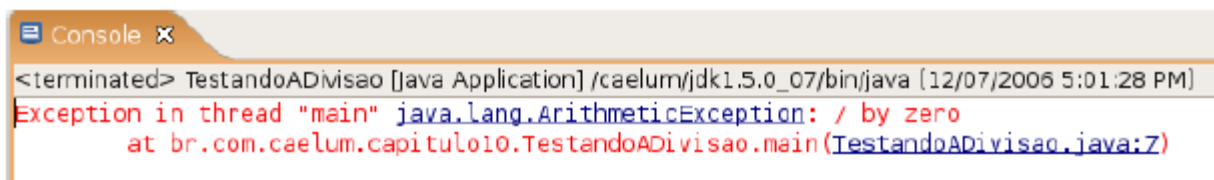
Repare que, a partir do momento que uma exception foi caught (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

11.3 – Exceções de Runtime mais comuns

Que tal tentar dividir um número por zero? Será que a JVM 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 " + i);
    }
}
```

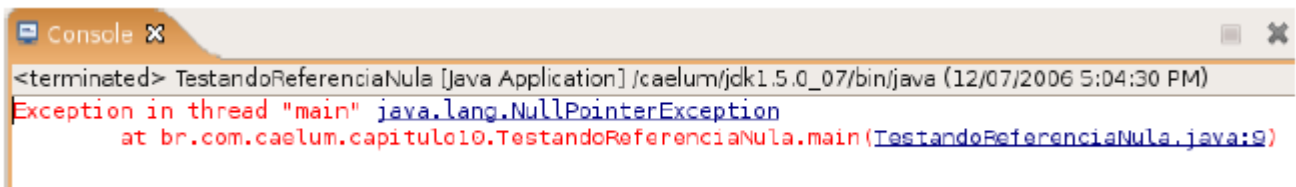
Tente executar o programa acima. O que acontece?



```
<terminated> TestandoADivisao [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 5:01:28 PM)
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at br.com.caelum.capitulo10.TestandoADivisao.main(TestandoADivisao.java:7)
```

```
public class TestandoReferenciaNula {
    public static void main(String args[]) {
        Conta c = null;
        System.out.println("Saldo atual " + c.getSaldo());
    }
}
```

Tente executar este programa. O que acontece?



```
<terminated> TestandoReferenciaNula [Java Application] /caelum/jdk1.5.0_07/bin/java (12/07/2006 5:04:30 PM)
Exception in thread "main" java.lang.NullPointerException
    at br.com.caelum.capitulo10.TestandoReferenciaNula.main(TestandoReferenciaNula.java:9)
```

Repare que um `ArrayIndexOutOfBoundsException` ou um `NullPointerException` poderia ser facilmente evitado com o `for` corretamente escrito ou com `ifs` que checariam os limites da array. Outro caso em que também ocorre tal tipo de exceção é quando um cast errado é feito (veremos mais pra frente). Em todos os casos, tais problemas provavelmente poderiam ser evitados pelo programador. É por esse motivo que o Java não te obriga a dar o `try/catch` nessas exceptions e chamamos essas exceções de unchecked. Em outras palavras, o compilador não checa se você está tratando essas exceções.

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, já que provavelmente o melhor a se fazer é deixar a JVM encerrar (ou apenas a Thread em questão).

11.4 – 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 a quem chama o método ou construtor a tratar essa exceção. Chamamos esse tipo de exceção de checked, pois o compilador checará se ela está sendo devidamente tratada, diferente das anteriores, conhecidas como unchecked.

Um exemplo interessante é o de abrir um arquivo para leitura, onde pode ocorrer o erro do arquivo não existir (veremos como trabalhar com arquivos em outro capítulo, **não** se preocupe com isto agora):

```
class Teste {
    public static void metodo() {
        new java.io.FileInputStream("arquivo.txt");
    }
}
```

O código acima não compila e o compilador avisa que é necessário tratar o FileNotFoundException que pode ocorrer:

```
Teste.java:3: unreported exception java.io.FileNotFoundException; must be caught
or declared to be thrown
    new java.io.FileReader("arquivo.txt");
    ^
1 error
```

Para compilar e fazer o programa funcionar, temos duas maneiras que podemos tratar o problema. 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.FileInputStream("arquivo.txt");
    } catch (java.io.FileNotFoundException e) {
        System.out.println("Nao foi possível abrir o arquivo para leitura");
    }
}
```

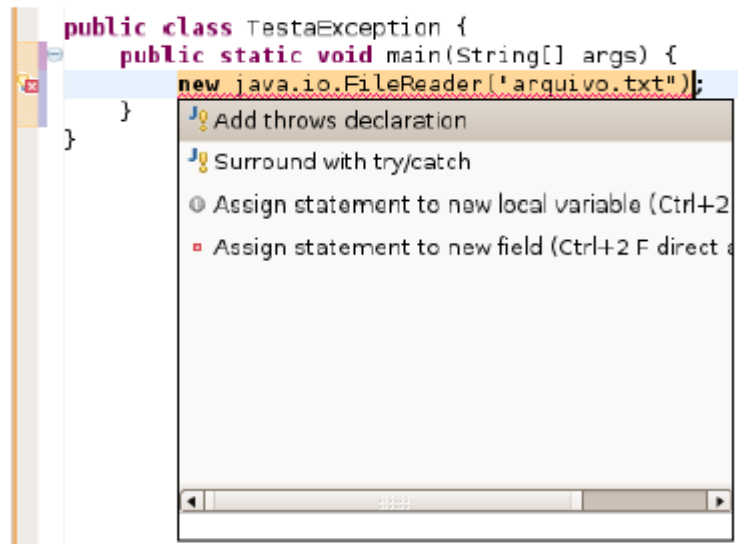
A segunda forma de tratar esse erro, é delegar ele para quem chamou o nosso método, isto é, passar para a frente.

```
public static void metodo() throws java.io.FileNotFoundException {
    new java.io.FileInputStream("arquivo.txt");
}
```

No Eclipse é bem simples fazer tanto um try/catch como um throws:
Tente digitar esse código no eclipse:

```
public class TestaException {
    public static void main(String[] args) {
        new java.io.FileInputStream("arquivo.txt");
    }
}
```

O Eclipse vai reclamar :



E você tem duas opções:

1) Add throws declaration, que vai gerar:

```
public class TestaException {
    public static void main(String[] args) throws FileNotFoundException {
        new java.io.FileInputStream("arquivo.txt");
    }
}
```

2) Surround with try/catch, que vai gerar:

```
public class TestaException2 {
    public static void main(String[] args) {
        try {
            new java.io.FileInputStream("arquivo.txt");
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

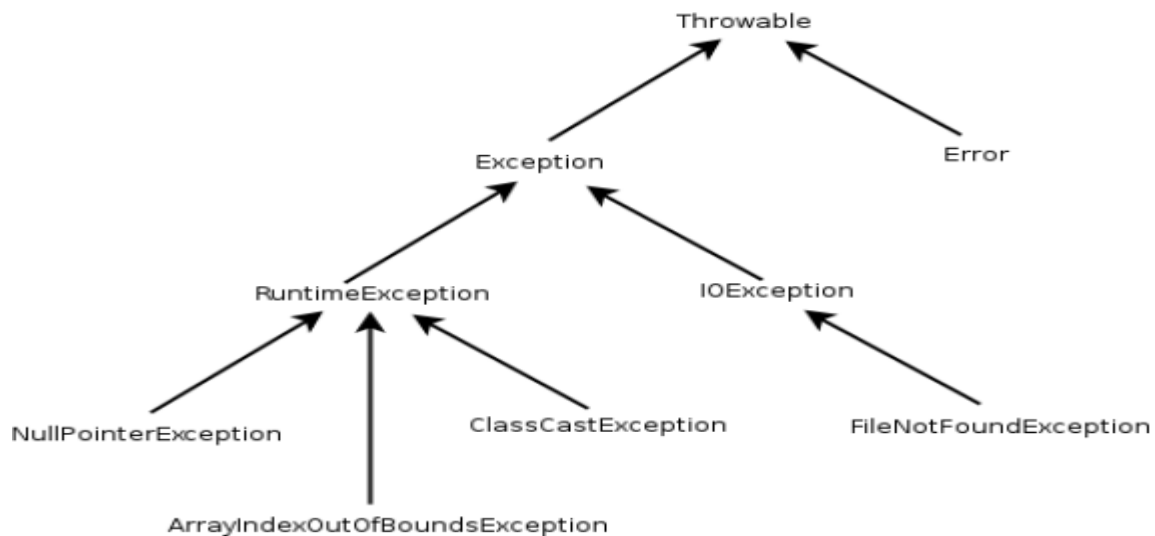
No início, existe uma grande tentação de sempre passar o problema pra frente para outros o tratarem. Pode ser que faça sentido, dependendo do caso, mas não até o main, por exemplo. Acontece que quem tenta abrir um arquivo sabe como lidar com um problema na leitura. Quem chamou um método no começo do 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 determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação

àquele erro. Enquanto não for o momento, você provavelmente vai preferir delegar a responsabilidade para o método que te invocou.

11.5 – Um pouco da grande família THROWABLE

Uma pequena parte da Família_rowable:



11.6 – 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 {
    // ..
}

```

3) Você pode, também, escolher tratar algumas exceções e declarar as outras no 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 às vezes, facilita a leitura e a documentação do seu código.

11.7 – Lançando exceções

Lembre-se do método saca da nossa classe Conta. Ele devolve um boolean caso consiga ou não sacar:

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

Podemos, também, lançar uma Exception, o que é extremamente útil. Dessa maneira, resolvemos o problema de alguém poder esquecer de fazer um if no retorno de um método. A palavra chave **throw**, que está no imperativo, lança uma Exception. Isto é bem diferente de throws, que está no presente do indicativo, e que apenas avisa da possibilidade daquele método lançá-la, obrigando o outro método que vá utilizar deste de se preocupar com essa exceção em questão.

```
void saca(double valor) {
    if (this.saldo < valor) {
        throw new RuntimeException();
    } else {
        this.saldo -= valor;
    }
}
```

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 saca(double valor) {
    if (this.saldo < valor) {
        throw new IllegalArgumentException();
    } else {
        this.saldo -= 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 a melhor escolha quando um argumento sempre é inválido como, por exemplo, números negativos, referências nulas, etc). Para pegar esse erro, não usaremos um if/else e sim um try/catch, porque faz mais sentido já que a falta de saldo é uma exceção:

```
Conta cc = new ContaCorrente();
cc.deposita(100);
try {
```

```

        cc.saca(100);
    } catch (IllegalArgumentException e) {
        System.out.println("Saldo Insuficiente");
    }

```

Podíamos melhorar ainda mais e passar para o construtor da `IllegalArgumentException` o motivo da exceção:

```

void saca(double valor) {
    if (this.saldo < valor) {
        throw new IllegalArgumentException("Saldo insuficiente");
    } else {
        this.saldo -= valor;
    }
}

```

O método `getMessage()` definido na classe `Throwable` (mãe de todos os tipos de erros e exceptions) vai retornar a mensagem que passamos ao construtor da `IllegalArgumentException`.

```

try {
    cc.saca(100);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

```

11.8 – O que colocar dentro do try

Imagine que vamos sacar dinheiro de diversas contas:

```

Conta cc = new ContaCorrente();
cc.deposita(100);
Conta cp = new ContaPoupanca();
cp.deposita(100);
// sacando das contas:
cc.saca(50);
System.out.println("consegui sacar da corrente!");
cp.saca(50);
System.out.println("consegui sacar da poupança!");

```

Podemos escolher vários lugares para colocar try/catch:

```

try {
    cc.saca(50);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
System.out.println("consegui sacar da corrente!");

try {
    cp.saca(50);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

```



```

}
System.out.println("consegui sacar da poupança!");

```

Essa não parece uma opção boa, pois a mensagem "consegui sacar" será impressa mesmo que o catch seja acionado. Sempre que temos algo que depende da linha de cima para ser correto, devemos agrupá-lo no try:

```

try {
    cc.saca(50);
    System.out.println("consegui sacar da corrente!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

try {
    cp.saca(50);
    System.out.println("consegui sacar da poupança!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

```

Mas há ainda uma outra opção: imagine que, para o nosso sistema, uma falha ao sacar da conta poupança deve parar o processo de saques e nem tentar sacar da conta corrente. Para isso, agruparíamos mais ainda:

```

try {
    cc.saca(50);
    System.out.println("consegui sacar da corrente!");
    cp.saca(50);
    System.out.println("consegui sacar da poupança!");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

```

O que você vai colocar dentro do try influencia muito a execução do programa! Pense direito nas linhas que dependem uma da outra para a execução correta da sua lógica de negócios.

11.9 – Criando sua própria exceção

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, que sejam úteis de alguma forma. Vamos criar a nossa:

Voltamos para o exemplo das Contas, vamos criar a nossa Exceção de SaldoInsuficienteException:

```

public class SaldoInsuficienteException extends RuntimeException {
    SaldoInsuficienteException(String message) {
        super(message);
    }
}

```

Em vez de lançar um `IllegalArgumentException`, vamos lançar nossa própria exception, com uma mensagem que dirá “Saldo Insuficiente”:

```
void saca(double valor) {
    if (this.saldo < valor) {
        throw new SaldoInsuficienteException("Saldo Insuficiente," +
            "tente um valor menor");
    } else {
        this.saldo -= valor;
    }
}
```

E, para testar, crie uma classe que deposite um valor e tente sacar um valor maior:

```
public static void main(String[] args) {
    Conta cc = new ContaCorrente();
    cc.deposita(10);
    try {
        cc.saca(100);
    } catch (SaldoInsuficienteException e) {
        System.out.println(e.getMessage());
    }
}
```

Podemos transformar essa `Exception` de unchecked para checked, obrigando a quem chama esse método a dar try-catch, ou throws:

```
public class SaldoInsuficienteException extends Exception {
    SaldoInsuficienteException(String message) {
        super(message);
    }
}
```

11.10 – Para saber mais: 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. É interessante colocar algo que é imprescindível de ser executado, caso o que você queria fazer tenha dado certo, ou não. O caso mais comum é o de liberar um recurso no finally, como um arquivo ou conexão com banco de dados, para que possamos ter a certeza de que aquele arquivo (ou conexão) vá ser fechado, mesmo que algo tenha falhado no decorrer do código.

No exemplo a seguir, o bloco finally será sempre executado, independentemente de tudo ocorrer bem ou de acontecer algum problema:

```
try {
    // bloco try
} catch (IOException ex) {
    // bloco catch 1
} catch (SQLException sqllex) {
    // bloco catch 2
}
```

```
} finally {  
    // bloco que será sempre executado, independente  
    // se houve ou não exception e se ela foi tratada ou não  
}
```

Há também, no Java 7, um recurso poderoso conhecido como try-with-resources, que permite utilizar a semântica do finally de uma maneira bem mais simples.