

Conteúdo

Introdução a Java.....	1
História (breve).....	2
Extensões.....	4
Estrutura Básica da Linguagem Java.....	6
Primeiro Programa Java.....	7
Outros Aplicativos Java: Uso de variáveis.....	9
Conceito de memória, Operadores e Tipos de dados.....	10
Estruturas de controle – Seleções.....	11
Estruturas de controle – repetições.....	12
Repetição controlada por sentinela.....	13
Instrução break e continue.....	13
Labeled loops.....	14
Arrays – Vetores e Matrizes.....	14
Declarando um array unidimensional(vetor):.....	14
Leitura/escrita/manipulação: usado para se percorrer o vetor:.....	15
Declarando um array bidimensional (matriz ou Arrays de Arrays):.....	15
Leitura/escrita/manipulação: usado para se percorrer a matriz:.....	15
JOptionPane – Comando de entrada e saída.....	15
Mensagens - showMessageDialog().....	16
Entrada de dados – showInputDialog().....	17
Confirmação – showConfirmDialog().....	17
Escolha de opções – showOptionDialog().....	18
Orientação a Objetos - Classes e Métodos.....	19
Classes objetos, métodos e variáveis de instância.....	19
Declarando uma classe com um método e instanciando um objeto de uma classe ..	22
Declarando um método com um parâmetro.....	23
Variáveis de instancia e atributos.....	24

Escopo de tempo de vida.....	24
Modificadores.....	26
Tipos primitivos versus tipos por referência	28
Construtores.....	29
Métodos com retorno	30
Observações sobre classes.....	31
Observações sobre métodos.....	34
Sobrecarga de métodos.....	34
Métodos get() e set().....	35
Listas de argumentos de comprimento variável	37
A clausula This	37
Variáveis de instancia final.....	38
Mais sobre abstração de dados e encapsulamento.....	38
Controlando acesso a membros - pacotes.....	39
Enumerações.....	39
Orientação a Objetos – Relacionamento entre Classes	42
Herança.....	42
Palavra reservada super.....	44
Utilizando atributos e métodos da classe mãe/superclasse.....	45
Polimorfismo	46
Classes e métodos abstratos.....	48
Clausula <i>instanceOf</i>	49
Downcasting.....	49
Sombreamento.....	50
Sobrecarga	51
Sobre-escrita	51
Interfaces	52
Erros e Exceções.....	53
Tipos de Exceção.....	54

Erros.....	55
Falhas.....	55
Exceções de Contingência	55
Exceções verificadas e não-verificadas.....	55
Tratamento de Exceções.....	57
Criando Exceções	58
Try-Finally.....	61
Genéricos.....	62
Classes genéricas – um exemplo	63
Tipos brutos	64
Coleções.....	65
Listas.....	66
ArrayList	66
Linkedlist.....	67
Algoritmos de coleção	67
Stack(Pilha).....	68
Queue(Fila).....	69
Conjuntos.....	69
Mapas	69
Exercícios resolvidos	70
Concorrência – Threads.....	87
Classe Thread.....	88
Prioridades de thread e agendamento de thread.....	92
Criando e executando threads	92
Colocando threads para dormir.....	95
Exercícios Complementares de Fixação	96
Apêndice 01 - Garbage collector	115
Coletor de lixo e o método finalize	115
Apêndice 02 – Saída Formatada	116

Formatando a saída com printf	116
Parâmetros de formato.....	116
Imprimindo datas e horas	117
Largura e precisão de campos	117
Apêndice 03 - Os 10 mandamentos do bom programador Java.....	118
Apêndice 04 - Separação de Responsabilidades e Encapsulamento.....	122
Apêndice 05 – Declarações e controle de Acesso	124
Identificadores legais.....	124
Convenção de código	124
Classes e interfaces.....	124
Métodos.....	124
variáveis.....	125
constantes	125
Apêndice 06 – palavras reservadas	126
Modificadores de acesso	126
Modificadores de classes, variáveis ou métodos	126
Controle de fluxo dentro de um bloco de código.....	127
Tratamento de erros	127
Controle de pacotes	128
Primitivos.....	128
Variáveis de referência.....	128
Retorno de um método	128
Palavras reservadas não utilizadas	128
Literais reservados	128

Introdução a Java

Atualmente Java é uma plataforma de programação madura e estabelecida - não só uma linguagem de programação como também um grande conjunto de bibliotecas e ferramentas. Está sendo amplamente usada em sistemas empresariais computadorizados e aplicativos, além de ganhar cada vez mais projeção em sistemas incorporados (Embedded Systems).

Java é usado como um título geral que engloba a linguagem de programação Java, as ferramentas e o ambiente do kit de desenvolvimento e todas as API's e bibliotecas de classes que são distribuídas como padrão. Até certo ponto todos esses elementos são inseparáveis, portanto aprender Java significa conhecer todos eles.

Java fornece uma linguagem de programação orientada a objetos. Atualmente, o desenvolvimento orientado a objetos é amplamente usado, já que resolve vários dos problemas que surgem durante o desenvolvimento de software, fornecendo soluções viáveis e práticas.

Não pense que “ser orientado a objetos” é como balançar uma varinha mágica que vai resolver todos os seus problemas de desenvolvimento de software. O que esse recurso vai lhe dar é uma estrutura conceitual e uma abordagem de desenvolvimento de software que se baseia em um conhecimento sólido do que funciona melhor. Java personifica bastante essa experiência.

A programação, principalmente a orientada a objetos, se baseia na identificação e no trabalho com abstrações. Um abstração permite que um conceito ou ideia seja expressa e, em seguida, usada repetidamente sem que todos os detalhes tenham de ser considerados.

Muitas abstrações básicas são usadas ampla e repetidamente, portanto não há motivo para recriá-las sempre.

O uso de abstrações não se resume às que vem com biblioteca Java, o programador tem que encontrar, aprender e usar outras bibliotecas de classe apropriadas que possam ser adicionadas ao ambiente de desenvolvimento ou, até mesmo, ele próprio construir novas bibliotecas. Encontrar e construir

abstrações é a chave para um desenvolvimento de sistemas bem sucedidos, principalmente no desenvolvimento orientado a objetos. Não saber usar e construir abstrações é realmente inadequado, e vai levar a programas mal projetados e instáveis.

História (breve)

Em 1991, na Sun Microsystems, foi iniciado o Green Project, o berço do Java, uma linguagem de programação orientada a objetos. Os mentores do projeto eram Patrick Naughton, Mike Sheridan, e James Gosling. O objetivo do projeto não era a criação de uma nova linguagem de programação, mas antecipar e planejar a “próxima onda” do mundo digital. Eles acreditavam que, em algum tempo, haveria uma convergência dos computadores com os equipamentos e eletrodomésticos comumente usados pelas pessoas no seu dia-a-dia. Para provar a viabilidade desta ideia, 13 pessoas trabalharam arduamente durante 18 meses. No verão de 1992 eles emergiram de um escritório de Sand Hill Road, no Menlo Park, com uma demonstração funcional da ideia inicial. O protótipo se chamava *7 (lê-se “StarSeven”), um controle remoto com uma interface gráfica touchscreen. Para o *7, foi criado um mascote, hoje amplamente conhecido no mundo Java, o Duke. O trabalho do Duke no *7 era ser um guia virtual ajudando e ensinando o usuário a utilizar o equipamento. O *7 tinha a habilidade de controlar diversos dispositivos e aplicações. James Gosling especificou uma nova linguagem de programação para o *7. Gosling decidiu batizá-la de “Oak”, que quer dizer carvalho, uma árvore que ele podia observar quando olhava pela sua janela.

O próximo passo era encontrar um mercado para o *7. A equipe achava que uma boa ideia seria controlar televisões e vídeo por demanda com o equipamento. Eles construíram uma demonstração chamada de MovieWood, mas infelizmente era muito cedo para que o vídeo por demanda bem como as empresas de TV a cabo pudessem viabilizar o negócio. A ideia que o *7 tentava vender, hoje já é realidade em programas interativos e também na televisão digital. Permitir ao telespectador interagir com a emissora e com a programação em uma grande rede de cabos, era algo muito visionário e estava muito longe do

que as empresas de TV a cabo tinham capacidade de entender e comprar. A ideia certa, na época errada.

Entretanto, o estouro da internet aconteceu e rapidamente uma grande rede interativa estava se estabelecendo. Era este tipo de rede interativa que a equipe do *7 estava tentando vender para as empresas de TV a cabo. E, da noite para o dia, não era mais necessário construir a infra-estrutura para a rede, ela simplesmente estava lá. Gosling foi incumbido de adaptar o Oak para a internet e em janeiro 1995 foi lançada uma nova versão do Oak que foi rebatizada para Java. A tecnologia Java tinha sido projetada para se mover por meio das redes de dispositivos heterogêneos, redes como a internet. Agora aplicações poderiam ser executadas dentro dos navegadores nos Applets Java e tudo seria disponibilizado pela internet instantaneamente. Foi o estático HTML dos navegadores que promoveu a rápida disseminação da dinâmica tecnologia Java. A velocidade dos acontecimentos seguintes foi assustadora, o número de usuários cresceu rapidamente, grandes fornecedores de tecnologia, como a IBM anunciaram suporte para a tecnologia Java.

Desde seu lançamento, em maio de 1995, a plataforma Java foi adotada mais rapidamente do que qualquer outra linguagem de programação na história da computação. Em 2004 Java atingiu a marca de 3 milhões de desenvolvedores em todo mundo. Java continuou crescendo e hoje é uma referência no mercado de desenvolvimento de software. Java tornou-se popular pelo seu uso na internet e hoje possui seu ambiente de execução presente em navegadores, mainframes, sistemas operacionais, celulares, palmtops, cartões inteligentes etc. Em 1997 a Sun Microsystems tentou submeter a linguagem a padronização pelos órgãos ISO/IEC e ECMA, mas acabou desistindo. Java ainda é um padrão de fato, que é controlada através da JCP Java Community Process. Em 13 de novembro de 2006, a Sun lançou a maior parte do Java como Software Livre sob os termos da GNU General Public License (GPL). Em 8 de maio de 2007 a Sun finalizou o processo, tornando praticamente todo o código Java como software de código aberto, menos uma pequena porção da qual a Sun não possui copyright.

Fonte: wikipedia

Extensões

Java é dividido em três grandes extensões, que englobam outras várias. Não há necessidade de saber todas elas. Aprenda aqueles que melhor atendem as suas necessidades.

As três grandes : JSE, JEE e JME

O J2SE (Java 2 Standard Edition) ou *Java SE* é uma ferramenta de desenvolvimento para a plataforma Java. Ela contém todo o ambiente necessário para a criação e execução de aplicações Java, incluindo a máquina virtual Java (JVM), o compilador Java, as APIs do Java e outras ferramentas utilitárias.

A plataforma Java EE (J2EE) inclui toda a funcionalidade existente na plataforma Java SE mais todas as funcionalidades necessárias para o desenvolvimento e execução de aplicações em um ambiente corporativo.

Java Platform, Micro Edition, Java ME, ou ainda J2ME, é uma tecnologia que possibilita o desenvolvimento de software para sistemas e aplicações embebidas ou embarcados, ou seja, toda aquela que roda em um dispositivo de propósito específico, desempenhando alguma tarefa que seja útil para o dispositivo.

Também temos outras extensões como:

Java Card é uma tecnologia que permite que pequenos aplicativos (*applets*) baseados em plataforma Java sejam executados com segurança em *smart cards* e dispositivos similares com limitações de processamento e armazenamento, como o *Java Ring*.

Java Database Connectivity ou JDBC é um conjunto de classes e interfaces (API) escritas em Java que fazem o envio de instruções SQL para qualquer banco de dados relacional; Api de baixo nível e base para api's de alto nível; Amplia o que você pode fazer com Java; Possibilita o uso de bancos de dados já instalados; Para cada banco de dados há um driver JDBC que pode cair em quatro categorias.

JavaServer Pages (JSP) é uma tecnologia utilizada no desenvolvimento de aplicações para Web, similar às tecnologias Active Server Pages (ASP)

da Microsoft ou PHP. Por ser baseada na linguagem de programação Java, tem a vantagem da portabilidade de plataforma, que permite a sua execução em diversos sistemas operacionais, como o Windows da Microsoft, Unix e Linux. Esta tecnologia permite ao desenvolvedor de páginas para Internet produzir aplicações que acessem o banco de dados, manipulem arquivos no formato texto, capturem informações a partir de formulários e capturem informações sobre o visitante e sobre o servidor.

Nesse curso iremos aprender sobre Java Básico, ou a Extensão JSE, englobando a estrutura inicial da linguagem Java. Iremos utilizar o conceito de orientação a objetos que é muito importante, como já descrito nessa introdução, passando por classes, métodos, herança, polimorfismo entre outras e também iremos ver uma biblioteca muito utilizada e fundamental, a Collection.

Notas do Autor:

Lembre-se sempre de fazer os exercícios, ler a apostila e se desafiar. Apenas com isso você ganhara experiência, conhecimento e domínio sobre o conteúdo passado durante esse treinamento. Lembre-se Estudar é fundamental e seu esforço será reconhecido.

Bibliografias utilizadas no material:

Sun Certified Programmer for Java® Platform, SE6 Study Guide. **Raposa**, Richard F.

Java - Como programar. **Deitel**, Harvey M.;

Java Collections. **Zukowski**, John;

Desenvolvendo Software em Java. **Winder**, Russel & **Roberts**, Graham

UML – Guia do Usuario. **Booch**, Grady **Rumbaugh**, James **Jacobson**, Ivar

Para dúvidas e problemas temos diversos fóruns. A comunidade de Programação é bem solidária e sempre vão te ajudar. Recomendo os fóruns GUI , JavaFree.org e DevMedia. Para qualquer dúvida referente ao conteúdo, material e exercícios, disponibilizo-me no email: biasonlucky@hotmail.com. Eu sou Lucas Biason, instrutor e autor desse material, muito obrigado e um bom estudo!

“Nossa vida é desperdiçada em detalhes... Simplifique, simplifique.” – Henry David Thoreau

Estrutura Básica da Linguagem Java

Objetivo: *Conhecer a estrutura básica da programação Java, controle de fluxos e variáveis.*

Programas Java consistem em partes chamadas de classes. As classes incluem partes chamadas de métodos que realizam tarefas e retornam informações ao concluir. Os programadores podem criar cada parte de que precisam para formar programas em Java. Entretanto, a maioria dos programadores Java tira proveito das ricas coleções de classes existentes nas bibliotecas de classes Java, que também são conhecidas como APIs do Java ou Java APIs (Application Programming Interfaces).

Portanto, há dois aspectos para aprender o “mundo” Java: 1) a própria linguagem em si, de modo que você possa programar suas próprias classes e 2) as classes nas extensas bibliotecas de classe Java.

Nota do Autor:

Utilize uma abordagem de blocos de construção para criar programas. Evite reinventar a roda – utilize partes existentes onde for possível. Chamada de reutilização de software, essa prática é fundamental para a programação orientada a objetos.

Ao programar em Java, em geral, você utilizará os seguintes blocos de construção: classes e métodos de bibliotecas de classes e métodos que você mesmo cria, e classes e métodos que outros criam e tornam disponíveis para você.

A vantagem de criar suas próprias classes e métodos é que você sabe como eles funcionam e pode examinar o código Java. As desvantagens são o consumo de tempo e o esforço potencialmente complexo que são necessários.

Nota do Autor:

Utilizar classes e métodos da API do Java em vez de escrever suas próprias versões pode melhorar o desempenho de programas, por que eles são cuidadosamente escritos para executar de modo eficiente. Essa técnica diminui o tempo de desenvolvimento dos programas e melhora a portabilidade de programas, por que esses são incluídos em cada implementação Java.

O Java é uma linguagem de programação muito poderosa. Os programadores experientes, às vezes sentem-se orgulhosos por criarem algum uso exótico, distorcido e complexo de uma linguagem. Essa pratica de programação é pobre.

Escreva seus programas Java de uma maneira simples e direta. Leia a documentação da versão do Java que você está utilizando, Consulte-a frequentemente para certificar-se de que está ciente da rica coleção de recursos Java e de que está utilizando esses recursos corretamente.

Seu computador e compilador são bons professores. Se, depois de ler cuidadosamente o manual de documentação do Java, você não estiver seguro sobre como um recurso do Java funciona, experimente para ver o que acontece. Estude cada erro ou mensagem de advertência/aviso que surge durante a compilação de programas e corrija os programas eliminar essas mensagens.

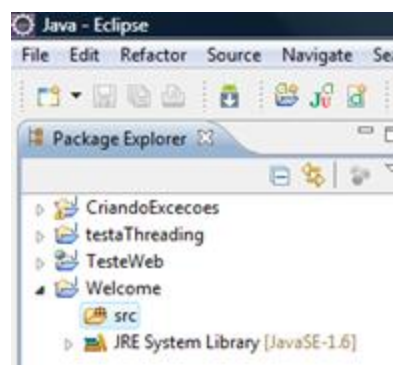
Primeiro Programa Java

Iremos utilizar a IDE Eclipse para desenvolver e compilar nossos programas Java. Para isso, ao abrir a IDE iremos clicar em File/New/Project. Vamos escolher Java Project, clicar em “next” e escolher um nome para nosso programa. Vamos chamar de Welcome (Bem-Vindo) e clicar em Finish.


Um novo projeto irá aparecer na aba mais a esquerda da IDE. Chamado Welcome, contendo duas pastas: src , a qual iremos colocar nossos pacotes, classes e afins durante o curso. E JRE System Library, que são as API's do Java.

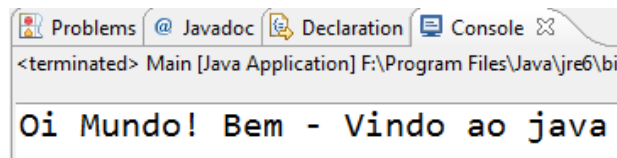
Iremos clicar com o botão direito. New/Class. Escolhemos o nome Main e finish.

Agora iremos digitar o exemplo a seguir.



```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Oi Mundo! Bem - Vindo ao java");  
    }  
}
```

Quando clicarmos o botão executar , irá aparecer no Console, a nossa mensagem escrita:



Agora iremos examinar o código escrito.

`public class Main {` é uma declaração de classe. Cada Programa Java consiste em pelo menos uma declaração de classe que é definida pelo programador (você). A palavra chave `class` introduz uma declaração de classe em Java e é imediatamente seguida pelo nome da classe.

Nota do Autor:

Por convenção, sempre inicie o identificador do nome de uma classe com uma letra maiúscula e inicie cada palavra subsequente no identificador com uma letra maiúscula. Nomear suas classes dessa maneira torna sei programas mais legíveis.

`public static void main(String[] args) {` é a declaração de um método especial que torna a nossa classe executável (não se preocupe com a terminologia por enquanto, mais a frente iremos ver sobre classes e métodos).

`System.out.println("Oi Mundo! Bem - Vindo ao java");` é um código para imprimir textos na tela do console para o usuário. Tudo o que estiver entre parênteses será mostrado como foi escrito.

Outros Aplicativos Java: Uso de variáveis

Nosso próximo aplicativo lê dois inteiros digitados por um usuário no teclado e calcula a soma dos valores e exibe o resultado.

Para a leitura dos dados utilizaremos uma classe da API do java chamada **Scanner**. Para isso devemos importar elas em nosso projeto, utilizando a instrução **import**. Como no código a seguir:

```
import java.util.Scanner;

public class Addition {

    public static void main(String[] args) {

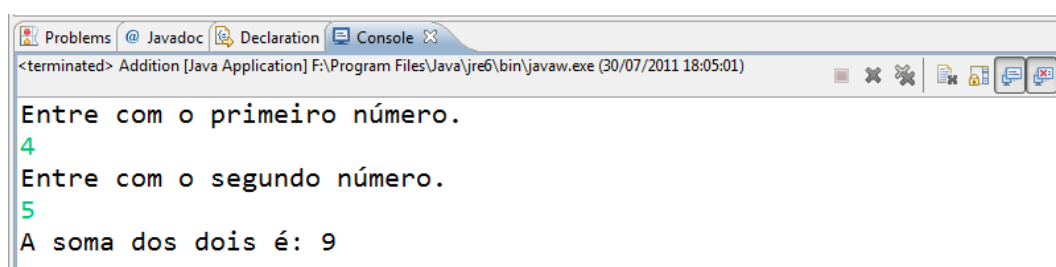
        Scanner input = new Scanner(System.in);

        System.out.println("Entre com o primeiro número.");
        int numero1 = input.nextInt();
        System.out.println("Entre com o segundo número.");
        int numero2 = input.nextInt();

        int soma = numero1 + numero2;

        System.out.println("A soma dos dois é: " + soma);
    }
}
```

Nesse exemplo temos instruções novas a serem vistas. Na terceira linha temos a declaração de uma variável no tipo **int** chamada *numero1* e a ela atribuímos um código diferente : **input.nextInt()**; , aqui usamos a funcionalidade da classe **Scanner** para capturar o próximo inteiro que o usuário digitar e atribuímos seu valor na variável. Fazendo isso para *numero1* e *numero2* criamos uma variável chamada *soma* para receber a soma. Até fácil né?



```
<terminated> Addition [Java Application] F:\Program Files\Java\jre6\bin\javaw.exe (30/07/2011 18:05:01)

Entre com o primeiro número.
4
Entre com o segundo número.
5
A soma dos dois é: 9
```

Nota do Autor:

Por convenção, identificadores de nomes de variáveis iniciam com uma letra minúscula e cada palavra no nome depois da primeira palavra inicia com uma letra maiúscula. Por exemplo: *primeiroNumero*.

Escolher nomes de variáveis significativos ajuda um programa a ser **autodocumentado** (isto é, pode-se entender o programa simplesmente lendo-o em vez de ler manuais ou comentários).

Conceito de memória, Operadores e Tipos de dados

Os nomes de variáveis como as do exemplo anterior correspondem as posições na memória do computador. Cada variável tem um **nome**, um **tipo** e um **valor**.

Para executar operações aritméticas com nossas variáveis podemos contar com os operadores aritméticos a seguir:

Adição	+	Ex: $n1 + n2$
Subtração	-	Ex: $n1 - n2$
Multiplicação	*	Ex: $n1 * n2$
Divisão	/	Ex: $n1/n2$
Resto	%	Ex: $n1 \% n2$

As expressões aritméticas em java devem ser escritas na forma de linha reta para facilitar inserir programas no computador.

Os parênteses são utilizados para agrupar termos em expressões Java. Isso porque o java aplica os operadores em expressões aritméticas em uma sequência precisa determinada pelas regras de precedência de operadores.

Essas regras permitem que o Java aplique os operadores na ordem correta. Dessa forma:

$Media = a + b + c / 3$ - errado

$Media = (a + b + c) / 3$ – correto, isso porque a divisão tem precedência mais alta que a adição. Use os parênteses, sempre nesses casos. Se os parênteses forem omitidos, apenas o valor de c será dividido por 3 e não o conjunto $a+b+c$.

Igual	==	X == y
Diferente	!=	X != y
Maior	>	X > y
Menor	<	X < Y
Maior ou igual	>=	X >= y
Menor ou igual	<=	X <= y

Como veremos mais a frente, uma condição é uma expressão que pode ser verdadeira ou falsa.

As condições nas instruções if podem ser formatadas utilizando os operadores de igualdade (== e !=) e os operadores relacionais (>, <, >= e <=).

Na tabela ao lado vemos os operadores de condições.

Estruturas de controle – Seleções

Os programas utilizam instruções de seleção para escolher entre cursos alternativos de ações. Por exemplo, suponha que a nota de aprovação de um exame seja 60. Usamos uma seleção simples:

Se a nota do aluno é maior ou igual a 60
Imprima "Aprovado"

```
if (notaAluno >= 60)
    System.out.println("Aprovado");
```

A instrução if de uma única seleção realiza uma ação indicada somente quando a condição é true; caso contrario, a ação é pulada. A instrução de seleção dupla if...else permite que o programador especifique uma ação a realizar quando a condição é verdadeira e uma ação diferente quando a condição é false.

Se a nota do aluno é maior ou igual a 60
Imprima "Aprovado"
Senão
Imprima "Reprovado"

```
if (notaAluno >= 60)
    System.out.println("Aprovado");
else
    System.out.println("Reprovado");
```

O java fornece o operador condicional (?:) que pode ser utilizado no lugar de uma instrução if...else. Esse é o único operador ternário do Java.

Variável = (condição) ? valor_se_verdadeira : valor_se_falsa

```
System.out.println( (notaAluno >= 60) ? "Aprovado" : "Reprovado");
```

```
if (notaAluno >= 90)
    System.out.println("A");
else if (notaAluno >= 80)
    System.out.println("B");
else if (notaAluno >= 70)
    System.out.println("C");
else if (notaAluno >= 60)
    System.out.println("D");
else
    System.out.println("F");
```

Um programa pode testar múltiplos casos colocando if...else dentro de outras if..else, como no exemplo ao lado.

A instrução if normalmente espera somente uma instrução no seu corpo. Para incluir várias, inclua o uso de chaves ({ e })

```
if (notaAluno >= 60)
    System.out.println("Aprovado");
else{
    System.out.println("Reprovado");
    System.out.println("Você precisa fazer " +
        "esse curso novamente");
}
```

Outra instrução de múltipla seleção muito usada é a switch (apelidada de switch case). Ela seleciona uma variável e define casos onde o valor da variável é igual ao determinado, se for, ele executa ações determinadas, se não passa para o próximo caso. Ao final, o programador pode definir um tratamento caso o valor da variável não seja nenhum dos avaliados.

```
switch(n){
case 0 :
    System.out.println("zero");
case 1 :
    System.out.println("um");
case 2 :
    System.out.println("dois");
case 3 :
    System.out.println("três");
default :
    System.out.println("nenhum dos casos");
}
```

Estruturas de controle – repetições

Uma instrução de repetição permite ao programador especificar que um programa deve repetir uma ação enquanto alguma condição permanecer verdadeira. Temos três formas de fazer um loop:

Utilizando While (Enquanto):

```
Enquanto (condição){  
    instruções  
}
```

```
while(product <= 100){  
    product = 3 * product;  
}
```

Utilizando for:

```
for (inicio; fim; passo){  
    instruções  
}
```

```
for (int i=0 ; i<= 100; i++){  
    System.out.println(i);  
}
```

Utilizando do:

```
do{  
    instruções  
}while(condição)
```

```
do{  
    product = 3 * product;  
}while(product <= 100);
```

Repetição controlada por sentinela

O usuário, ou digita um valor que se pede ou digita um determinado valor, ao digitar o segundo, o laço termina.

```
int n;  
do{  
    System.out.println("Digite um número positivo ou -1 para sair");  
    n = input.nextInt();  
}while(n != -1);  
}
```

*enquanto o usuário não digitar -1 para sair, o programa irá executar o laço e as instruções. Nesse caso não existe nº definido de vezes.

Instrução break e continue

Ambas alteram o fluxo de controle.

- **Break** é usado para parar. Quando numa estrutura switch, while, for ou do, no momento que ele é acionado, o programa termina o laço de repetição e pula para o restante do código após o laço, mesmo que a condição ainda seja verdadeira.

- **Continue** é usado para pular instruções. Quando usado em while, do...while ou for ele pula as instruções restantes no corpo do loop e prossegue com a próxima iteração do loop.

```
do{
    System.out.println("Digite um número positivo ou -1 para sair");
    n = input.nextInt();
    if (n == -1) break;
    if (n <= 0) continue;
    System.out.println("você digitou: "+n);
}while(n != -1);
```

*No exemplo acima, se o numero digitado for -1, ele sai no laço. Se for negativo, não mostra a mensagem, se for positivo, ele mostra o numero.

Labeled loops

Podemos também dar nomes (labels) para os loops. Muito útil quando se tem loops múltiplos, um interno ao outro.

```
primeiro: for (int i =0; i <100; i++){
    System.out.println("primeiro: "+i);
    segundo: for (int j =0; j <100; j++){
        if(j==10) break primeiro;
        if( j% 2 == 0) continue segundo;
        System.out.println("\tsegundo: "+j);
    }
}
```

Arrays – Vetores e Matrizes

Um array é um grupo de variáveis (elementos) que contém valores que são todos do mesmo tipo. Um array é uma referencia (visto mais a frente). Os elementos de um array podem ser de tipos primitivos ou tipos por referencia.

Declarando um array unidimensional(vetor):

Podemos declara arrays de três jeitos:

```
String [] Nomes = new String[5];
```

O primeiro jeito e o segundo criam um vetor de 5 palavras (Strings).

```
String [] nomes;
nomes = new String[5];
```

```
int[] numeros = {1,2,3,4,5};
```

O ultimo jeito cria um vetor de inteiros já preenchido e determinado com tamanho 5.

Leitura/escrita/manipulação: usado para se percorrer o vetor:

```
for(int i=0; i<nomes.length; i++){  
    //codigo  
}
```

Declarando um array bidimensional (matriz ou Arrays de Arrays):

```
String [][] Nomes2 = new String[5][5];  
  
String [][] Nomesirregular = new String[5][];  
Nomesirregular[0] = new String[5];  
Nomesirregular[1] = new String[4];  
Nomesirregular[2] = new String[3];  
Nomesirregular[3] = new String[2];  
Nomesirregular[4] = new String[7];  
  
int [][] numeros2 = {  
    {1,2,3,4,5},  
    {1,2,3},  
    {1,2,3,4,5,6}  
};
```

Os arrays bidimensionais costumam ser utilizados para representar tabelas de valores que consistem nas instruções dispostas em linhas e colunas.

Em Java dizemos que a estrutura matriz (array bidimensional) é um vetor de vetor. Como podemos ver ao lado. Podemos criar uma matriz quadrada(1º modo), e matrizes irregulares (2º e 3º modo).

Leitura/escrita/manipulação: usado para se percorrer a matriz:

```
for(int linha=0; linha <numeros2.length; linha++){  
    for(int coluna=0; coluna<numeros2.length; coluna++){  
        //codigos  
    }  
}  
//percorre as linhas
```

JOptionPane – Comando de entrada e saída

Outra forma de entrada e saída de dados em Java é utilizando interfaces gráficas. Começaremos com as JOptionPane, as caixas de diálogos. Para utiliza-las é preciso importar a biblioteca **javax.swing.JOptionPane**.

Temos quatro tipos de caixas de diálogos. E em cada uma podemos escolher o tipo de ícone que irá aparecer:

Pergunta



JOptionPane.QUESTION_MESSAGE

Informação



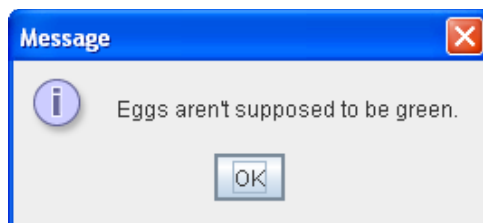
JOptionPane.INFORMATION_MESSAGE

Aviso		JOptionPane.WARNING_MESSAGE
Erro		JOptionPane.ERROR_MESSAGE

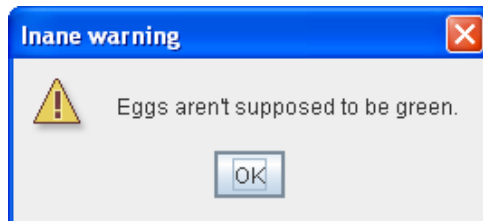
Mensagens - showMessageDialog()

Apresenta uma mensagem e não retorna nenhum tipo de valor. Sua sintaxe é:

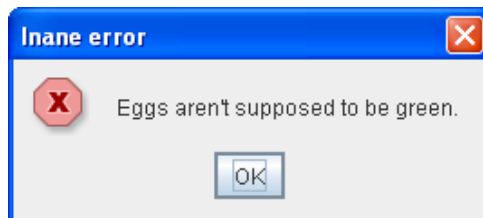
JOptionPane.showMessageDialog(null,"mensagem","titulo","ícone");



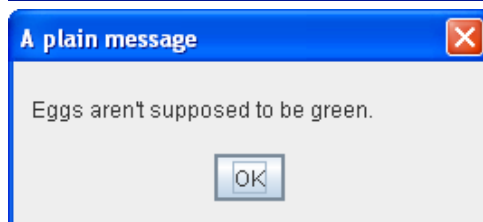
```
//PADRÃO ou INFORMATION_MESSAGE
JOptionPane.showMessageDialog(
    null,
    "Eggs are not supposed to be green.",
    "Message",
    JOptionPane.INFORMATION_MESSAGE);
```



```
//aviso
JOptionPane.showMessageDialog(
    null,
    "Eggs are not supposed to be green.",
    "Inane warning",
    JOptionPane.WARNING_MESSAGE
);
```



```
//erro
JOptionPane.showMessageDialog(
    null,
    "Eggs are not supposed to be green.",
    "Inane error",
    JOptionPane.ERROR_MESSAGE
);
```

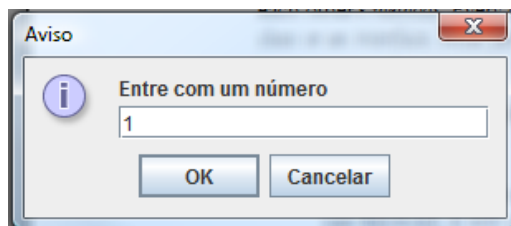


```
//sem ícone
JOptionPane.showMessageDialog(
    null,
    "Eggs are not supposed to be green.",
    "A plain message",
    JOptionPane.PLAIN_MESSAGE
);
```

Entrada de dados – `showInputDialog()`

Além de emitir uma mensagem, permite a entrada de texto, sempre retorna uma string que é o conteúdo digitado pelo usuário. Sua Sintaxe é:

Variável = `JOptionPane.showInputDialog (null, "mensagem", "titulo", "ícone");`



```
String a = JOptionPane.showInputDialog(  
    null,  
    "Entre com um número",  
    "Aviso",  
    JOptionPane.INFORMATION_MESSAGE);
```

*Como ela sempre retorna uma Strings, para converter o retorno para inteiro ou real usamos: `int numero = Integer.parseInt(<código>);` e `float numero = Float.parseFloat(<código>);` onde <código> é algum código que retorne uma string que representa um numero.

Confirmação – `showConfirmDialog()`

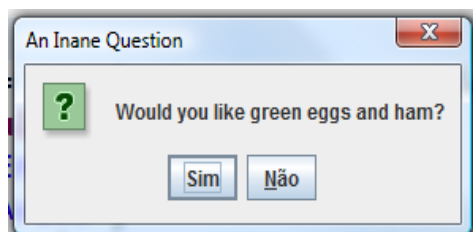
Além de emitir uma mensagem, possibilita ao usuário responder a uma pergunta por meio dos botões “yes,no,cancel”. Uma vez apresentada na tela, o usuário escolhe uma opção e dependendo do botão clicado, é gerado um valor inteiro pertencente a classe `JOptionPane`:

Yes -> 0;

No -> 1;

Cancel -> 2;

Esses valores são utilizados para reconhecer qual dos botões foi clicado pelo usuário.



```
int n = JOptionPane.showConfirmDialog(  
    null,  
    "Would you like green eggs and ham?",  
    "An Inane Question",  
    JOptionPane.YES_NO_OPTION);
```

Sua sintaxe é:

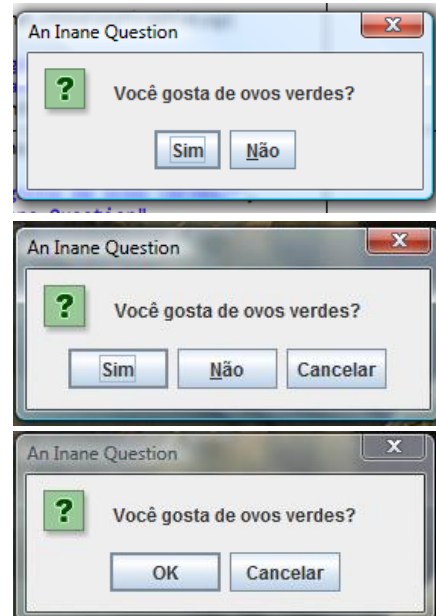
Variável = `JOptionPane.showConfirmDialog(null, "mensagem", " titulo", <botoes>, <ícone>);`

Onde <botões> é um inteiro de 0 até 2 que difere em três tipos as caixas de confirmação:

```
n = JOptionPane.showConfirmDialog(
    null,
    "Você gosta de ovos verdes?",
    "An Inane Question",
    JOptionPane.YES_NO_OPTION);

n = JOptionPane.showConfirmDialog(
    null,
    "Você gosta de ovos verdes?",
    "An Inane Question",
    JOptionPane.YES_NO_CANCEL_OPTION);

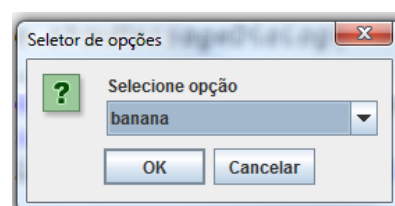
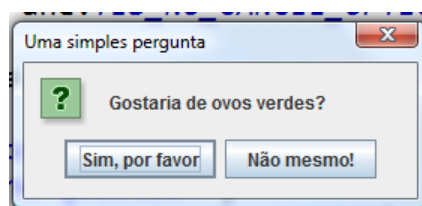
n = JOptionPane.showConfirmDialog(
    null,
    "Você gosta de ovos verdes?",
    "An Inane Question",
    JOptionPane.OK_CANCEL_OPTION);
```



Escolha de opções – showOptionDialog()

São complexas a posnto de combinar todos os recursos das anteriores retornando um inteiro referente ao index da opção selecionada. Sua sintaxe é:

Variável = JOptionPane(null,"mensagem", "titulo", <botoes>,<ícone>, null, array de objetos, seleção padrão);



```
Object[] options = {"Sim, por favor", "Não mesmo!"};
n = JOptionPane.showOptionDialog(null,
    "Gostaria de ovos verdes?",
    "Uma simples pergunta",
    JOptionPane.YES_NO_OPTION,
    JOptionPane.QUESTION_MESSAGE,
    null, //do not use a custom Icon
    options, //the titles of buttons
    options[0]); //default button title

Object selecao =
    JOptionPane.showInputDialog(
        null,
        "Selecione opção",
        "Seleção de opções",
        JOptionPane.QUESTION_MESSAGE,
        null,
        new Object[]
        { "banana", "kiwi", "morango" },
        "banana");
```

“Nada pode ter valor sem ser um objeto útil” – Karl marx

Orientação a Objetos - Classes e Métodos

Objetivo: iniciar as técnicas de programação orientada a objetos. Utilizando classes, métodos e atributos. Para a construção de programas estruturados apropriadamente para ser gerenciáveis.

Classes objetos, métodos e variáveis de instância

Iniciemos com uma analogia simples para ajudar a entender classes e seu conteúdo. Suponha que você queira guiar um carro e fazê-lo andar mais rápido pisando no acelerador. O que deve acontecer antes que você possa fazer isso? Bem, antes de você poder dirigir um carro, alguém tem de projetar o carro. Em geral, um carro 'nasce' com os desenhos de engenharia, semelhantes às plantas utilizadas para projetar uma casa. Esses desenhos de engenharia incluem o projeto de um pedal acelerador para aumentar a velocidade do carro. O pedal 'oculta' os complexos mecanismos que realmente fazem o carro ir mais rápido, assim como o pedal de freio 'oculta' os mecanismos que diminuem sua velocidade e a direção 'oculta' os mecanismos que mudam a direção do carro. Isso permite que as pessoas com pouco ou nenhum conhecimento de como os motores funcionam dirijam um carro facilmente.

Infelizmente, você não pode guiar os desenhos de engenharia de um carro. Antes de poder guiar um carro, ele deve ser construído a partir dos desenhos de engenharia que o descrevem. Um carro pronto terá um pedal acelerador real para fazer o carro andar mais rápido, mas até isso não é suficiente - o carro não acelerará por conta própria, então o motorista deve pressionar o pedal acelerador.

Para realizar uma tarefa em um programa é necessário um **método**. O método descreve os mecanismos que realmente realizam suas tarefas. O método oculta de seu usuário as tarefas complexas que ele realiza, assim como o pedal acelerador de um carro oculta do motorista os complexos mecanismos que fazem o carro andar mais rápido.

Em Java, primeiro criamos uma unidade de programa chamada **classe** para abrigar um método, assim como os desenhos de engenharia do carro

abrigam o projeto de um pedal acelerador. Em uma classe, você fornece um ou mais métodos que são projetados para realizar as tarefas da classe. Por exemplo, uma classe que representa uma conta bancária poderia conter um método para fazer depósitos de dinheiro em uma conta, outro para fazer saques e um terceiro para perguntar qual é o saldo atual.

!Assim como você não pode dirigir um desenho de engenharia de um carro, você não pode 'dirigir' uma classe.

Assim como alguém tem de construir um carro a partir de seus desenhos de engenharia antes de você poder realmente guiar o carro, você deve construir um **objeto** de uma classe antes de fazer um programa realizar as tarefas que a classe descreve como fazer!

Essa é uma razão de o Java ser conhecido como uma linguagem de programação orientada a objetos.

Ao dirigir um carro, o ato de pressionar o acelerador envia uma mensagem para o carro realizar uma tarefa - isto é, fazer o carro andar mais rápido. De maneira semelhante, você envia mensagens para um objeto - cada mensagem é conhecida como uma **chamada de método** e instrui um método do objeto a realizar sua tarefa.

Nota do Autor:

Encapsulamento é um mecanismo que assegura que tudo tenha lados interno e externo bem definidos. O lado interno fica protegido do que ocorre no lado externo. Os mecanismos internos ficam ocultos e não precisamos saber de sua existência. Portanto, a visão externa e a interface apresentam a abstração, ocultando os detalhes internos. Geralmente o encapsulamento é considerado um mecanismo de ocultação de informações, por razões óbvias.

Até aqui, utilizamos a analogia do carro para introduzir classes, objetos e métodos. Além das capacidades de um carro, ele também tem muitos **atributos**, como cor, número de portas, quantidade de gasolina no tanque, velocidade atual e total de quilômetros percorrido (isto é, a leitura do

odômetro). Como as capacidades do carro, esses atributos são representados como parte do projeto de um carro em seus diagramas de engenharia.

Quando você dirige um carro, esses atributos estão sempre associados com o carro. Cada carro mantém seus próprios atributos.

Por exemplo, cada carro sabe a quantidade de gasolina que há no seu tanque, mas não sabe quanto há no tanque de outros carros. De maneira semelhante, um objeto tem atributos que são portados com o objeto quando ele é utilizado em um programa.

Dica: *classes* são como formas.
Objetos são como os bolos feitos nessas

Dica: *métodos* são as ações que um objeto de uma classe pode fazer.

Atributos são as características.

Esses atributos são especificados como parte da classe do objeto. Por exemplo, um objeto conta bancária tem um atributo saldo que representa a quantidade de dinheiro na conta. Cada objeto conta bancária sabe o saldo da conta que ele representa, mas não sabe os saldos de outras contas no banco. Os atributos são especificados pelas variáveis de instância da classe.

Formalizando...

Classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, métodos ou operações, relacionamentos e semântica.

Métodos definem o comportamento da classe. São funções.

Atributos ou variáveis de instancia. São valores da classe. Seu escopo abrange toda a classe.

Objeto uma entidade com existência física que pertence a um determinado conjunto de entidades afins (classe).

Instancia de objeto a instancia é criação do objeto na memoria.

Uma chamada de método executa os comandos do método.

Declarando uma classe com um método e instanciando um objeto de uma classe

Classes contêm métodos e atributos. Muitas vezes elas precisam comunicar-se entre si. A orientação a objetos permite que os objetos de uma classe recebam informações de outra classe e retornem valores a ela ou outra classe. Basicamente, para uma classe utilizar um método de outra classe é necessário a criação de uma **instancia de objeto**:

Classe_nome objeto_nome = new Classe_nome();

Agora vamos criar um novo projeto e nele uma nova classe chamada Oi. Ela terá um método chamado mensagem, como na figura a seguir:

```
public class Oi {
```

```
    public void mensagem(){
        System.out.println("eu sou um método");
    }
```

↓
método

Não é uma classe executável, pois não contém o método main(). Mas pode ser acessada por uma classe executável.

Depois vamos criar uma nova classe chamada OiTeste, que instancia um objeto da classe Oi e convoca (chamada de método) o método mensagem().

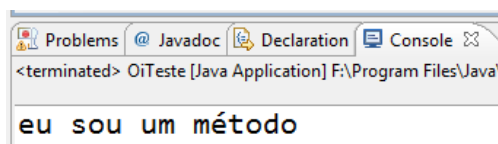
```
public class OiTeste {
```

```
    public static void main(String[] args) {
        Oi ola = new Oi();
        ola.mensagem();
    }
```

→ Cria um objeto Oi

→ Chama o método "mensagem" do objeto oi

O resultado vemos:



```
<terminated> OiTeste [Java Application] F:\Program Files\Java\
eu sou um método
```

Por convenção, os nomes de métodos começam com letra minúscula. Assim como o nome dos atributos. A próxima palavra do nome fica com letra maiúscula. Ex:

```
public void gerarMensagem()
```

Declarando um método com um parâmetro

Em alguns casos (muitos) os métodos precisam receber algum (ns) parâmetro(s) para realizar as suas tarefas. Nesse próximo exemplo iremos criar um livro de notas de um curso. Crie uma classe chamada LivroNotas e um métodos mostraMensagem. Ele receberá uma Strings, ou uma palavra como parâmetro.

```
public class LivroNotas {  
    public void mostraMensagem( String nomeCurso ) {  
        System.out.println("bem vindo ao livro de notas de "+ nomeCurso);  
    }  
}
```

Vamos criar uma nova classe executável para testar a nossa classe. Nela pedimos para o usuário entrar com o nome do seu curso. E depois mandamos o objeto executar o método mostraMensagem. Repare que temos que passar uma Strings para esse método e também que não precisamos saber como funciona internamente o método graças ao principio do encapsulamento.

```
import java.util.Scanner;  
  
public class LivroTeste {  
  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        LivroNotas meuLivroNotas = new LivroNotas();  
        System.out.println("Por favor entre com o nome do curso");  
        String nomeCurso = input.nextLine();  
        System.out.println();  
  
        meuLivroNotas.mostraMensagem(nomeCurso);  
    }  
}
```

Nota do Autor:

Normalmente os objetos são criados com new. Uma exceção é um literal de Strings que está entre aspas, como "hello". Os literais de Strings são referencias a objetos Strings que são criados implicitamente pelo Java.

Ocorrerá um erro de compilação se o numero de argumentos em uma chamada de método não corresponder ao numero de parâmetros na declaração de método.

Nota do Autor:

Ocorrerá um erro de compilação se os tipos dos argumentos em uma chamada de método não forem consistentes com os tipos dos parâmetros correspondentes na declaração do método.

Variáveis de instancia e atributos

As variáveis declaradas no corpo de um método particular são conhecidas como variáveis locais e só podem ser utilizadas nesse método. Quando esse método terminar, os valores de suas variáveis locais são perdidos.

Uma classe normalmente consiste em um ou mais métodos que manipulam os atributos que pertencem a um objeto particular da classe. Os atributos são representados como variáveis em uma declaração de classe. Essas variáveis são chamadas de campos e são declaradas dentro de uma declaração de classe, mas fora dos corpos das declarações de método da classe. Quando cada objeto de uma classe mantém sua própria cópia de um atributo, o campo que representa o atributo também é conhecido como uma variável de instância - cada objeto (instância) da classe tem uma instância separada da variável na memória.

Escopo de tempo de vida

Para entender como variáveis são gerenciadas precisamos introduzir dois conceitos importantes:

Escopo – compreende as partes de um programa em que uma determinada variável pode ser usada.

Tempo de vida – determina quando uma variável ou objeto é criado e destruído.

Toda variável tem um escopo, a parte de um programa em que ela é declarada e onde pode ser usada. Comandos de um programa no mesmo escopo podem usar a variável após sua declaração, enquanto comandos externos a esse escopo não podem usá-la de forma alguma.

Cada método e comando composto define um escopo local. Uma variável declarada em um escopo local só pode ser usada por comandos nesse escopo (isto é, no método ou comando composto). Portanto, um método pode ter variáveis locais que só comandos existentes dentro desse método podem usar. Da mesma forma, um comando composto pode ter variáveis locais que só podem ser usadas nesse comando – o corpo de um método é essencialmente um comando composto, logo o método define um escopo por que o comando composto o define. Embora aparentemente declarados fora de um comando composto, parâmetros estão dentro do escopo do método, portanto são variáveis locais.

Escopos podem ser aninhados, logo podem existir dentro de outro escopo. Isso ocorre quando comandos compostos são usados dentro do corpo de um método. Qualquer variável declarada em um escopo pode ser usada em um escopo interno ou aninhado mas variáveis declaradas em um escopo interno só podem ser usadas dentro desse escopo. O uso de escopos aninhados fornece uma maneira de reduzir a região de um programa e que uma variável pode ser usada. Isso reduz o risco de a variável ser usada inadequadamente em uma parte do programa que não devia ter acesso a ela, portanto o escopo é um mecanismo que ajuda o programador a fazer um encapsulamento. Um bom princípio de programação que deve ser seguido é sempre limitar o escopo de uma variável ao mínimo possível.

Já que variáveis só podem ser usadas em seu escopo, só precisam existir quando estão no escopo. Graças a isso a linguagem pode assegurar que variáveis só sejam criadas quando necessárias, uma vez fora do escopo podem ser destruídas, e a memória pode ser reusada. Variáveis locais tem um tempo de vida igual ao de seu escopo.

Isso significa que sempre que um método é chamado um novo conjunto de variáveis locais e de parâmetros é criado e inicializado. Quando a chamada de método retorna, as variáveis saem do seu escopo e são destruídas, e portanto aquele conjunto específico de variáveis não pode ser usado novamente. Da próxima vez que o método for chamado um novo conjunto de variáveis será criado. É esse mecanismo que permite a recursão; cada chamada resulta na criação de novas variáveis. Chamadas recursivas diferentes não compartilham

variáveis. Esse princípio também é aplicado a comandos compostos, inclusive os usados em corpos de laço. Considere o trecho de código a seguir:

```
int x=0;
while( x < 10){
    int y = x;
    //aqui tanto x quanto y podem ser usados
}
//aqui x pode ser usado
//aqui y NÃO pode ser usado, já que a declaração de y
//ocorreu em um escopo interno que já foi fechado
```

A ideia de escopo não está limitada a variáveis. Na verdade, todos os nomes de um programa tem escopo, inclusive nomes de variáveis e métodos.

Nota do Autor:

Deixando mais claro: O escopo de uma declaração é a parte do programa que pode referenciar a entidade declarada pelo seu nome. As regras básicas:

1. O escopo de uma declaração de parâmetro é o corpo do método em que a declaração aparece (parâmetro).
2. O escopo de uma declaração de variável local vai do ponto em que declaração aparece até o final desse bloco (local).
3. O escopo de uma declaração de variáveis local que aparece na seção de inicialização do cabeçalho de uma instrução for é o corpo da instrução for com as outras expressões no cabeçalho. (instrução)
4. O escopo de um método ou campo de uma classe é o corpo inteiro da classe. Isso permite que métodos não-static de uma classe utilizem os campos e outros métodos da classe. (geral na classe (private) ou globais (public).

Modificadores

Neste exato momento você deve estar se perguntando para que servem aquelas palavra todas usadas na hora de escrever um método em uma classe ou mesmo uma variável em um objeto como muitas vezes viu em nossos exemplos de código. Pois bem, é sobre isso que falaremos agora, sobre os modificadores.

- **final:** impossibilita que uma classe seja estendida, que um método seja sobrescrito ou que uma variável seja reinicializada (constantes). (* sobrescrita e extensão serão vistos mais a frente)
- **abstract:** classe que não pode ser instanciada ou método que precisa ser implementado por uma subclasse não abstrata, muito usado quando nos referimos a uma classe que poderia ser apenas uma classificação de um objeto no mundo real.
- **static:** faz um método ou variável pertencer a classe ao invés de às instancias. Não há necessidade de criar um objeto, você pode chamar o método escrevendo o nome da classe, ponto, e o nome do método (com seus parâmetros caso haja).
- **private :** acesso apenas dentro da classe. Muito usado para permitir um baixo acoplamento. Desse modo nenhuma outra classe pode utiliza-lo.
- **protected:** acesso por classes no mesmo pacote e subclasse. (visto mais a frente em “Controlando acesso a membros – pacotes”).
- **public :** acesso liberado a qualquer classe. “todo mundo ver e acessa”.
- **transiente:** impede a serialização de campos. (serialização é vista mais a frente).
- **synchronized:** indica que um método só pode ser acessado por uma thread de cada vez. Esse modificador será revisto na parte que tratará de threads. (threads são vistas mais a frente).
- **strictfp:** usado na frente a um método ou classe para indicar que os números de ponto flutuante seguirão as regras de ponto flutuante em todas as expressões.
- **volatile:** indica que uma variável pode ser alterada durante o uso de threads.
- **native:** indica que o método está escrito em uma linguagem dependente de plataforma, como o C.

Nota do Autor:

Anteceda cada campo e cada declaração de método com um modificador de acesso.

Nota do Autor:

Variáveis de objetos quase nunca devem ser públicas.

Variáveis de objeto (quase) sempre devem ser privadas, para que possam ser usadas internamente, na classe em que são declaradas. Os métodos que compõem a interface são públicos para que possam ser usadas externamente a classe em que foram declaradas.

Tipos primitivos versus tipos por referência

Os tipos de dados em Java estão divididos em duas categorias - tipos primitivos e tipos por referência (às vezes de tipos não-primitivos). Os tipos primitivos são **boolean**, **byte**, **char**, **short**, **int**, **long**, **float** e **double**. Todos os tipos não-primitivos são tipos por referência, então as classes, que especificam os tipos de objetos, são tipos por referência.

Uma variável de tipo primitivo pode armazenar exatamente um valor de seu tipo declarado por vez. Por exemplo, uma variável `int` pode armazenar um número inteiro (como 7) por vez. Quando outro valor for atribuído a essa variável, seu valor inicial será substituído.

As variáveis de instância de tipo primitivo são inicializadas por padrão - as variáveis de tipos `byte`, `char`, `short`, `int`, `long`, `float` e `double` e são inicializadas como 0 (zero), e as variáveis de tipo `boolean` são inicializadas como `false`. Os programadores podem especificar seus próprios valores iniciais para variáveis de tipo primitivo. Lembre-se de que as variáveis locais não são inicializadas por padrão.

Os programas utilizam as variáveis de tipos por referência (normalmente chamados de referências) para armazenar as localizações de objetos na memória do computador. Diz-se que essas variáveis referenciam objetos no programa. Os objetos que são referenciados podem todos conter muitas variáveis de instância e métodos.

As variáveis de instância de tipo por referência são inicializadas por padrão com o valor `null` - uma palavra reservada que representa uma 'referência a nada'.

Construtores

Cada classe que você declara pode fornecer um construtor que pode ser utilizado para inicializar um objeto de uma classe quando o objeto for criado.

Declaração:

```
public <Nome da classe> ( <parâmetros caso necessario> ) {  
    //código  
}
```

No nosso exemplo do LivroNotas, criamos um atributo chamado nomeCurso do tipo Strings e um construtor que recebe um parâmetro de mesmo nome que o atributo e o inicializa-se.

```
public class LivroNotas {  
  
    //atributo da classe  
    private String nomeCurso;  
  
    //construtor inicializando a variavel nomeCurso  
    public LivroNotas(String nomeCurso){  
        this.nomeCurso = nomeCurso;  
    }  
  
    public void mostraMensagem() {  
        System.out.println("bem vindo ao livro de notas de "+ nomeCurso);  
    }  
}
```

! O Java requer uma chamada de construtor para todo objeto que é criado. A palavra-chave new chama o construtor da classe para realizar a inicialização. A chamada de construtor é indicada pelo nome da classe seguido por parênteses!

Por padrão, o compilador fornece um construtor padrão sem parâmetros em qualquer classe que não inclua explicitamente um construtor.

Ao declarar uma classe, você pode fornecer seu próprio construtor a fim de especificar a inicialização personalizada para objetos de sua classe.

Em nosso teste, passamos o parâmetro direto no construtor. Como podemos ver, aquele código depois do **new** até então estranho, nada mais é do

que o construtor da classe. Repare que todo o construtor deve ter o mesmo nome da classe e **não** retorna valores.

```
import java.util.Scanner;

public class LivroTeste {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.println("Por favor entre com o nome do curso");
        String nomeCurso = input.nextLine();
        //criando o objeto passando o parametro para o construtor
        LivroNotas meuLivroNotas = new LivroNotas(nomeCurso);
        System.out.println();

        meuLivroNotas.mostraMensagem();
    }
}
```

Nota do Autor:

Como métodos, os construtores também podem aceitar argumentos. Entretanto, uma diferença importante entre construtores e métodos é que os construtores não podem retornar valores, portanto não podem especificar um tipo de retorno (nem mesmo void).

Normalmente, os construtores são declarados public. Se uma classe não incluir um construtor, as variáveis de instância da classe são inicializadas como seus valores-padrão. Se um programador declarar qualquer construtor para uma classe, o Java não criará um construtor-padrão para essa classe.

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. Modificamos o nosso exemplo para retornar uma Strings (que é a mensagem) para a classe que a chamar (LivroTeste).

```
public class LivroNotas {  
  
    //atributo da classe  
    private String nomeCurso;  
  
    //construtor inicializando a variavel nomeCurso  
    public LivroNotas(String nomeCurso){  
        this.nomeCurso = nomeCurso;  
    }  
  
    public String mostraMensagem() {  
        return "bem vindo ao livro de notas de " + nomeCurso;  
    }  
  
}
```

Observe que no nosso teste, agora recebemos a mensagem retornada pelo método.

```
public class LivroTeste {  
  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        System.out.println("Por favor entre com o nome do curso");  
        String nomeCurso = input.nextLine();  
        //criando o objeto passando o parametro para o construtor  
        LivroNotas meuLivroNotas = new LivroNotas(nomeCurso);  
        System.out.println();  
  
        System.out.println( meuLivroNotas.mostraMensagem() );  
    }  
  
}
```

Observações sobre classes

Vamos criar uma classe chamada Conta. Sua responsabilidade será abstrair uma conta de banco. Ela tem atributos como dono, saldo, limite e certas ações/métodos como sacar dinheiro, depositar algo.

Lembre-se que uma classe deve se abster a ter somente atributos e métodos condizentes a sua responsabilidade.

Classes com muitas responsabilidades são grandes, e difícil de manter, programar e dar manutenção.

Por isso sempre crie classes com uma responsabilidade apenas e o mesmo vale para seus métodos. Métodos devem fazer apenas o necessário e estipulado em sua responsabilidade.

Por exemplo, o método saca da classe Conta faz apenas o ato de tirar dinheiro do saldo. O método deposita apenas aumenta o saldo. Viu? Apenas uma responsabilidade para cada método. Não é viável construir um método que saca, deposita e ainda consulta o saldo. Assim como não é viável eu criar uma classe que cuide abstrair uma conta bancaria e uma venda em um caixa de supermercado.

Veja em nosso exemplo.

```
public class Conta {  
  
    //atributos  
    int numero;  
    String dono;  
    double saldo;  
    double limite;  
    double salario;  
  
    //métodos  
    public boolean saca(double valor) {  
        if (this.saldo < valor) {  
            return false;  
        } else {  
            this.saldo = this.saldo - valor;  
            return true;  
        }  
    }  
  
    void deposita(double quantidade) {  
        this.saldo += quantidade;  
    }  
}
```

Veja, métodos que fazem apenas uma ação são fáceis de entender, programar e dar manutenção. E uma classe com métodos “limpos” e que fazem ações apenas condizente com sua responsabilidade também são classes fáceis de entender, programar e dar manutenção.

```
public class ContaTeste {  
    public static void main(String[] args) {  
        //criamos um nova conta  
        Conta minhaConta;  
        minhaConta = new Conta();  
  
        //definimos quem é o dono e seu saldo  
        //utilizando os atributos da conta  
        minhaConta.dono = "Lucas";  
        minhaConta.saldo = 2300.0;  
  
        //mostramos o saldo  
        System.out.println("Saldo atual: " + minhaConta.saldo);  
  
        // saca 150 reais  
        minhaConta.saca(150);  
        System.out.println("Saldo depois do saque: " + minhaConta.saldo);  
  
        // deposita 700 reais  
        minhaConta.deposita(700);  
        System.out.println("Saldo depois do deposito: " + minhaConta.saldo);  
  
        //eu não tenho 15 mil na conta  
        boolean conseguiu = minhaConta.saca(15000);  
        if (conseguiu){  
            System.out.println("Consegui sacar");  
        } else {  
            System.out.println("Não consegui sacar");  
        }  
        //eu tenho 100 reais na minha conta  
        conseguiu = minhaConta.saca(15000);  
        if (conseguiu){  
            System.out.println("Consegui sacar");  
        } else {  
            System.out.println("Não consegui sacar");  
        }  
    }  
}
```

Criamos um teste onde são chamados os métodos saca, deposita e testamos o que acontece caso sacamos uma quantia maior do que a que tem.

Nota do Autor:

Dê um nome a sua classe de modo que fique claro qual é seu proposito ou finalidade. O mesmo para métodos e variáveis.

Siga as convenções usadas em Java para dar nomes a classes, métodos e variáveis e use-as consistentemente. Além de ajuda-lo a ler seu código, vai ajudar outras pessoas a lê-lo.

Nota do Autor:

Composição: uma classe pode ter referencias a objetos de outras classes como membros, ou seja, um método pode chamar outros. Uma forma de reutilização de software é a composição em que uma classe tem como membros referencias a objetos de outras classes.

Observações sobre métodos

Os métodos permitem que o programador modularize um programa separando suas tarefas em unidades autocontidas. Eles devem estar limitados a realização de uma única tarefa bem definida e o nome do método deve expressar essa tarefa efetivamente. Três maneiras de se chamar um método:

- 1- Próprio nome : método(parâmetros...) – usado quando o método esta na mesma classe que essa chamada.
- 2- Variáveis de referencia: variável.método(parâmetro); - tipo mais comum de chamada de método.
- 3- Classe.método() : usado para métodos estáticos (vide mais a frente em modificadores)

Sobrecarga de métodos

Mencionamos que nomes devem ser únicos em um determinado escopo. Isso é realmente verdade para nomes de variáveis, mas não é bem assim quanto a nomes de métodos. Continua sendo verdade que todas as variáveis e métodos devem ser identificáveis de maneira exclusiva, mas os métodos, diferentemente das variáveis, não são identificados somente por seu nome. Na verdade, o identificador de um método é uma combinação do nome e da quantidade e tipo dos parâmetros, o que geralmente é chamado de assinatura do método. Isso significa que pode haver dois ou mais métodos com o mesmo nome, mais assinaturas diferentes em um mesmo escopo.

Esse recurso da linguagem é chamado de **sobrecarga**, e chamamos esses métodos de **métodos sobrecarregados**. O tipo de retorno NÃO é levado em consideração. Portanto os métodos sobrecarregados podem retornar diferentes tipos.

A classe a seguir ilustra o uso de sobrecarga. Um objeto dessa classe gera Strings em um formato adequado para algum outro programa processar.

```
public class Formatter {  
  
    private final StringBuilder buffer = new StringBuilder();  
    //método add sobre-carregado  
    public void add (final int i){  
        buffer.append("I"+i+" ");  
    }  
    public void add (final double d) {  
        buffer.append("R"+d+" ");  
    }  
    public void add (final char c) {  
        buffer.append(c);  
    }  
    public void add (final String s) {  
        buffer.append(s);  
    }  
    //retorna a String montada  
    public String toString(){  
        return buffer.toString();  
    }  
}
```

Métodos get() e set()

Os campos **private** (privados) de uma classe podem ser manipulados somente pelos métodos dessa classe.

Os métodos **set()** são comumente chamados de métodos modificadores, pois geralmente atribuem valores a variáveis de instancias.

Os métodos **get()** são comumente chamados de métodos de acesso, pois obtém(retorna a outras classes) o valor das variáveis de instancia.

Uma variável de instancia publica pode ser lida ou gravada por qualquer método que tem uma referencia a um objeto que contem a variável de instancia. Se uma variável de instancia for declarada **private**, um método **get** publico certamente permitira que outros métodos acessem essa variável, mas o método **get** pode controlar como o cliente pode acessar essa variável.

Portanto, embora os métodos **set** e **get** possam fornecer acesso a dados privados, o acesso é restrito pela maneira como os métodos foram implementados pelo programador.

Métodos **set** de uma classe podem retornar valores indicando que foram feitas tentativas de atribuir dados inválidos a objetos da classe.

Modificamos o nosso exemplo da conta para demonstrar o uso de get e set:

```
public class Conta {  
  
    //atributos  
    private int numero;  
    private String dono;  
    private double saldo;  
    private double limite;  
    private double salario;  
  
    public int getNumero() {  
        return numero;  
    }  
  
    public void setNumero(int numero) {  
        this.numero = numero;  
    }  
  
    public String getDono() {  
        return dono;  
    }  
  
    public void setDono(String dono) {  
        this.dono = dono;  
    }  
  
    public double getSaldo() {  
        return saldo;  
    }  
  
    public void setSaldo(double saldo) {  
        this.saldo = saldo;  
    }  
  
    public double getLimite() {  
        return limite;  
    }  
  
    public void setLimite(double limite) {  
        this.limite = limite;  
    }  
  
    public double getSalario() {  
        return salario;  
    }  
  
    public void setSalario(double salario) {  
        this.salario = salario;  
    }  
}
```


Listas de argumentos de comprimento variável

Os programadores podem criar métodos que recebem numero não especificado de argumento. O uso de ... após a declaração de tipo de um parâmetro define o argumento. Veja o exemplo:

```
public class Palavras {  
  
    //parametros variaveis são tratados como arrays  
    //como todos os membros de um unico tipo estipulado  
    public String montaLista(String ... palavras){  
        String str="";  
  
        //foreach é um for especial que segue o seguinte modelo:  
        //for( Tipo var : colecao ou arranjo)  
        for(String stringAtual : palavras){  
            str = str + " " + stringAtual;  
        }  
  
        return str;  
    }  
}
```

Desse modo, o método pode receber de zero até infinitos parâmetros. Zero? Sim, o uso desses pontos torna a passagem de parâmetro não só variável em número mas também opcional, ou seja, o método pode ser chamado sem a necessidade de passagem de parâmetros.

O for utilizado é o **foreach**, que também serve para qualquer coleção (visto mais a frente) e arrays (arranjos). Isso por que todos os parâmetros passados a esse método são colocado em um array do nome e tipo declarado.

É possível declarar outros parâmetros juntos ao argumento. Porem eles não podem ser argumentos, ou seja, devem ser 'normais' e o argumento deve vir após todos os parâmetros 'normais', ou seja, por ultimo.

A clausula This

Cada objeto pode acessar uma referencia a si próprio com a palavra-chave **this**. (não pode ser utilizado em métodos static

Variáveis de instancia final

O princípio de menor privilegio é fundamental para uma boa engenharia de software. O príncipe declara que deve ser concedido ao código somente a quantidade de privilegio e acesso que o código precisa para realizar sua tarefa designada e não mais que isso.

Algumas variáveis de instancia precisam ser modificadas e algumas não. Você pode utilizar a palavra-chave final para especificar o fato de que uma variável não é modificável (isto é, constante) e que qualquer tentativa de modificar é um erro:

Private final tipo constante_nome;

Embora as constantes possam ser inicializadas quando são declaradas, isso não é exigido. Constantes podem ser inicializadas por cada um dos construtores de classe.

Um campo final também deve ser declarado static se for inicializado na sua declaração. Depois que um campo final é inicializado na sua declaração, seu valor nunca pode mudar. Não é necessário criar uma cópia separada do campo para cada objeto da classe. Criar o campo static permite que todos os objetos da classe compartilham o campo final.

Mais sobre abstração de dados e encapsulamento

Normalmente, classes ocultam os detalhes de implementação dos seus clientes. Isso se chama ocultamento de informações.

O cliente se preocupa com a funcionalidade, não com o modo como essa funcionalidade é implementada. Esse conceito é conhecido como abstração de dados.

!Embora programadores talvez conheçam os detalhes da implementação de uma classe, eles não devem escrever código que depende desses detalhes. Isso permite a uma classe particular ser substituída por outra versão sem afetar o restante do sistema contanto que os serviços public da classe não mudam, o restante do sistema não é afetado!

O Java e o estilo de programação orientado a objetos elevam a importância dos dados. As principais atividades da programação em Java orientada a objetos são a criação de tipos e a expressão das iterações entre objetos desses tipos.

A noção de tipo de dados abstratos ADT – abstract data type - melhora o processo de desenvolvimento de programas, capturando duas noções: representação de dados e as operações que podem ser realizadas nesses dados. Programadores Java utilizam classes para implementar tipos de dados abstratos.

ADT garante a integridade de sua estrutura de dados interna. Os clientes não podem manipular essa estrutura de dados diretamente. Os clientes só podem realizar operações admissíveis sobre a representação de dados.

Controlando acesso a membros - pacotes

Os pacotes ajudam a gerenciar a complexidade dos componentes do aplicativo. Os pacotes também facilitam a reutilização de software permitindo que programas importem classes de outros pacotes. Outro benefício é que eles fornecem uma convenção para nomes únicos de classes, o que ajuda a evitar conflitos entre nomes e classes.

Para criar uma classe reutilizável:

- 1) Declare a classe public;
- 2) Declare **package** e o nome do pacote: **package pacote.NomeClasse;**
- 3) Importando a classe: **import pacote.NomeClasse;**

Enumerações

Um tipo enum básico define um conjunto de constantes representadas como identificadores únicos. Como ocorre com classes, todos os tipos enum são tipos por referência, o que significa que você pode referenciar um objeto de um tipo enum com uma referência.

Um tipo enum é declarado com uma declaração enum, uma lista separada por vírgulas de constantes enum - a declaração pode opcionalmente incluir outros componentes das classes tradicionais como construtores, campos e métodos.

Cada declaração enum declara uma classe enum com as seguintes restrições:

1. Tipos enum são implicitamente final porque declaram constantes que não devem ser modificadas.
2. Constantes enum são implicitamente static. Não há necessidade de criar um objeto enum.
3. Qualquer tentativa de criar um objeto de um tipo enum com um operador new resulta em um erro de compilação.

As constantes enum podem ser utilizadas em qualquer lugar em que constantes podem ser utilizadas, como nos rótulos case das instruções switch e para controlar instruções for aprimoradas.

Criamos um exemplo, o enum chamado AnimalType, por convenção gosto de colocar o sufixo “Type” ou “Tipo” nos nomes deles.

```
public enum AnimalType {  
  
    //declara constantes do tipo enum  
    caninos("Cães"),  
    felinos("Gatos"),  
    repteis("Répteis"),  
    aves("Aves"),  
    peixes("Peixes"),  
    ornitorinco("Ornitorinco");  
  
    private final String nome;  
  
    private AnimalType(String nome) {  
        this.nome = nome;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
}
```

Cada constante enum é opcionalmente seguida por argumentos que são passados para o construtor enum. Como os construtores que você viu nas classes, um construtor enum pode especificar qualquer número de parâmetros e pode ser sobrecarregado.

Neste exemplo, o construtor enum tem um parâmetro String, consequentemente cada constante enum é seguida por parênteses contendo um argumentos Strings.

```
public class TestaAnimais {

    public static void main(String[] args) {

        System.out.println( "All books :\n" );
        //imprime todos os animais em enum AnimalType
        for (AnimalType animal : AnimalType.values()){
            System.out.println(" 1- " + animal.getNome());
        }

    }

}
```

<terminated> TestaAnimais [Java Application] F:\Program Files\Java\jre6\bin\javaw.exe (02/08/2011 15:39:57)

All books :

- 1- Cães
- 1- Gatos
- 1- Répteis
- 1- Aves
- 1- Peixes
- 1- Ornitorinco

“Nunca diga que conhece inteiramente uma pessoa, até dividir uma herança com ela”. – Johann Kasper Lavater

Orientação a Objetos – Relacionamento entre Classes

Objetivo: relacionar as classes e seus métodos e atributos de forma a modelar sistemas reais e complexos utilizando a POO.

Herança

Continuamos nossa discussão sobre programação orientada a objetos, POO (OOP - object-oriented programming) introduzindo um de seus principais recursos - **herança**, que é uma forma de reutilização de software na qual uma nova classe é criada, absorvendo membros de uma classe existente e aprimorada com capacidades novas ou modificadas. Com a herança, os programadores economizam tempo durante o desenvolvimento de programa reutilizando software de alta qualidade testado e depurado. Isso também aumenta a probabilidade de um sistema ser implementado efetivamente.

Ao criar uma classe, em vez de declarar membros completamente novos, o programador pode designar que a nova classe deverá herdar membros de uma classe existente. Esta classe existente é chamada de **superclasse**, e a nova classe, de **subclasse**. (A linguagem de programação C++ refere-se à superclasse como a classe básica e a subclasse como a classe derivada.) Cada subclasse pode tornar-se a superclasse para futuras subclasses. Uma subclasse normalmente adiciona seus próprios campos e métodos. Portanto, uma subclasse é mais específica que sua superclasse e representa um grupo mais especializado de objetos. Em geral, a subclasse exibe os comportamentos de sua superclasse e comportamentos adicionais que são específicos à sua classe.

A **superclasse direta** é a superclasse a partir de qual a subclasse herda explicitamente. Uma superclasse indireta é qualquer superclasse acima da classe direta na hierarquia de classe, que define os relacionamentos de herança entre as classes. No Java, a hierarquia de classe inicia com a classe Object (no pacote Java.lang), que toda classe em Java direta ou indiretamente estende (ou 'herda de'). No caso da **herança simples**, uma classe é derivada de uma superclasse direta. O **Java**, ao contrário de C++, **não suporta herança múltipla** (que ocorre quando uma classe é derivada de mais de uma superclasse direta).

A experiência na criação de sistemas de software indica que quantidades significativas de código lidam com casos especiais intimamente relacionados. Quando os programadores estão preocupados com casos especiais, os detalhes podem obscurecer a visão geral. Com a programação orientada a objetos, os programadores se concentram nos aspectos comuns entre objetos no sistema em vez de nos casos especiais.

Distinguimos entre o relacionamento '**é um**' e o relacionamento '**tem um**'. '**É um**' representa a herança. Em um relacionamento '**é um**', um objeto de uma subclasse também pode ser tratado como um objeto de sua superclasse. Por exemplo, um carro é um veículo. Por contraste, '**tem um**' representa a **composição**. Em um relacionamento '**tem um**', um objeto contém uma ou mais referências de objeto como membros. Por exemplo, um carro tem uma direção (e um objeto carro tem uma referência a um objeto direção).

Novas classes podem herdar de classes em bibliotecas de classe. As organizações desenvolvem suas próprias bibliotecas de classe e tiram proveito de outras disponíveis no mundo. Algum dia, a maioria dos softwares novos provavelmente será construída a partir de **componentes reutilizáveis padronizados**, assim como os automóveis e a maioria dos hardwares de computadores é construída hoje. Isso facilitará o desenvolvimento de softwares mais poderosos, em maior número e mais baratos.

Lembrete do Autor!

Composição: uma classe pode ter referências a objetos de outras classes como membros, ou seja, um método pode chamar outros. Uma forma de reutilização de software é a composição em que uma classe tem como membros referências a objetos de outras classes.

Exemplo:

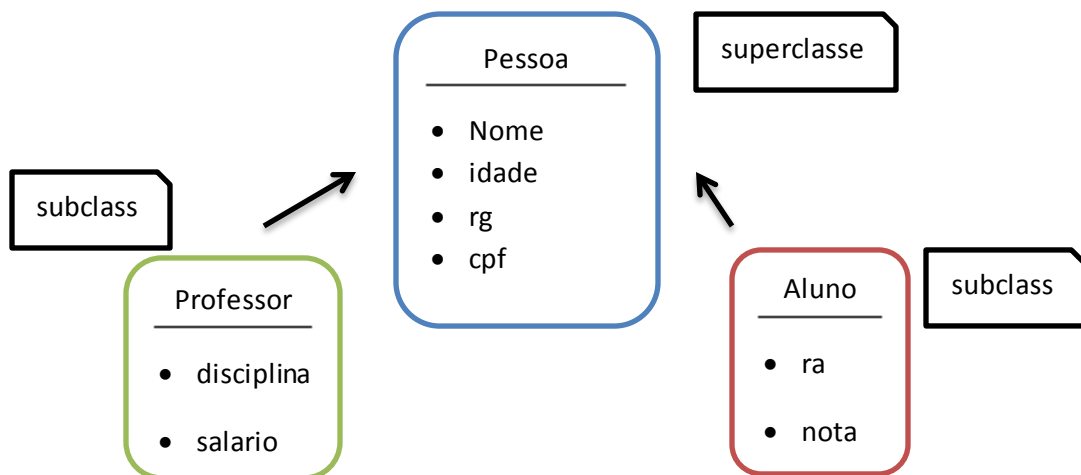
Professor

- Nome
- idade
- rg
- cpf
- disciplina
- salario

Aluno

- Nome
- idade
- rg
- cpf
- ra
- nota

Note que no exemplo, a classe Professor e a classe Aluno contêm os mesmos atributos e alguns diferenciados. Portanto podemos colocá-los em um grupo acima chamado Pessoa. A classe pessoa terá os atributos em comum.



Todo Aluno e Professor é uma Pessoa, eles tem os mesmos atributos de uma Pessoa. Note que mesmo não declarando os atributos nas classes Professor e Aluno, ele ainda tem esses atributos, isso porque vamos fazê-los herdar esses atributos da classe pessoa. E além dos atributos que Aluno e Professor herdarão de Pessoa, eles complementam as informações, deixam mais específica a classe, chamamos isso de **Especialização**. Do contrario, quando tenho atributos em comuns, e criamos a superclasse, fazemos uma **Generalização**.

Para declarar uma herança em Java utilizamos o comando **extends** <nome da classe mãe> após a declaração de classe. Veja nosso exemplo implementado:

```

public class Pessoa {
    String nome;
    String idade;
    String RG;
    String CPF;
}

public class Aluno extends Pessoa{
    int ra;
    float nota;
}

public class Professor extends Pessoa{
    String disciplina;
    float salario;
}
    
```

Palavra reservada super

Cada construtor de subclasse deve chamar implícita ou explicitamente seu construtor de superclasse para assegurar que as variáveis de instancia herdadas da superclasse sejam inicializadas adequadamente. Para isso utilizamos a palavra **super** seguida dos parâmetros do construtor da superclasse (caso haja).

Utilizando atributos e métodos da classe mão/superclasse

Utilizamos a palavra **super**, sempre que for necessário chamar algum método da superclasse:

```
super.metodo();
```

```
super.getAtributo();
```

```
super.setAtributo();
```

Nota do Autor

Copiar e colar códigos de uma classe para outra pode espalhar erros por múltiplos arquivos de código-fonte. Para evitar a duplicação de código (e possíveis erros), utilize a herança, em vez da abordagem 'copiar e colar', em situações em que você que uma classe absorva as variáveis de instancia e métodos de outra classe.

Com a herança, as variáveis de instância comuns e os métodos de todas as classes na hierarquia são declarados em uma superclasse. Quando as alterações são requeridas para esses recursos comuns, os desenvolvedores de software só precisam fazer as alterações na superclasse – as subclasses então herdam as alterações.

Sem a herança, as alterações precisariam ser feitas em todos os arquivos de código-fonte que contem uma copia do código em questão.

Podemos personalizar a nova classe para atender nossas necessidades incluindo membros adicionais e sobrescrevendo membros de superclasse. Fazer isso não exige que o programador de subclasses altere o código o código-fonte da superclasse.

! na etapa de projeto em um sistema orientado a objetos, o projetista frequentemente descobre que certas classes são proximamente relacionadas. O projetista deve fatorar as variáveis de instancia e método comuns e coloca-los em uma superclasse, especializando-as com capacidades além daquelas herdadas da superclasses!

Declarar uma subclasse não afeta o código-fonte da sua superclasse. A herança preserva a integridade da superclasse.

Polimorfismo

O **polimorfismo** permite ‘programar no geral’ em vez de ‘programar no específico’. O polimorfismo permite escrever programas que processam objetos que compartilham a mesma superclasse em uma hierarquia de classes como se todas fossem objetos da superclasse. Sem modificar o sistema, os programadores podem utilizar o polimorfismo para incluir tipos adicionais que não foram considerados quando o sistema foi criado.

Ex: Criando um novo projeto, vamos definir um sistema de hierarquia de um mini zoológico. Gato, Cachorro e cavalo são todos Animais e todos tem atributos e métodos em comum. No exemplo vamos nos focar no método **gritar()** (ou seja, fazer barulho). O barulho de um gato é diferente de um cachorro ou um cavalo, o grito de um cachorro é diferente de um gato e um cavalo e o grito de um cavalo é diferente de um gato e um cavalo. Veja as classes, nelas, Cavalo, Cachorro e Gato sobrescrevem o método herdado de Animal, **gritar()**, mostrando suas varias formas possíveis.

```
public class Animal {
    public void gritar(){
        System.out.print("");
    }
}

public class Gato extends Animal{
    @Override
    public void gritar(){
        System.out.print("Miau!!");
    }
}

public class Cachorro extends Animal{
    @Override
    public void gritar(){
        System.out.print("AuAuAu!!");
    }
}

public class Cavalo extends Animal{
    @Override
    public void gritar(){
        System.out.print("IRirii!");
    }
}
```

Nota do Autor

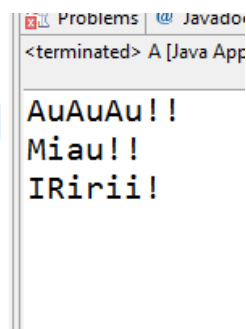
Sobrescrita de método é o ato de uma subclasse refazer o método herdado de uma superclasse. Ela apaga as funcionalidades desse método, definidas na superclasse, e transcreve a sua funcionalidade do seu jeito.

Com o polimorfismo, o mesmo nome e assinatura de método podem ser utilizados para fazer com que diferentes ações ocorram, dependendo do tipo de objeto em que o método é invocado.

- 1) O objeto "a" cria o objeto "b"

```
public class A {
    void façaAlgo(){
        Animal b;
        //sendo animal um cachorro
        b = new Cachorro();
        b.gritar();
        //sendo animal um gato
        b = new Gato();
        b.gritar();
        //sendo animal um cavalo
        b = new Cavalo();
        b.gritar();
    }
}
```

Resultado:



```
Problems | Java
<terminated> A [Java App
AuAuAu!!
Miau!!
IRirii!
```

- 2) O objeto "a" recebe o objeto "b" de um objeto "c"

```
void façaAlgoB(Animal c){
    this.b = c;
    b.gritar();
}
```

* repare que nesse caso fica impossível de saber de que animal será o grito. É um exemplo máximo para o polimorfismo, pois podemos ter qualquer tipo de grito nesse caso

- 3) O objeto "a" recebe o objeto "b" numa chamada de método

```
void façaAlgoC(Animal b){
    b.gritar();
}
```

Em Java, o polimorfismo se manifesta apenas em chamadas de métodos. Polimorfismo significa que uma chamada de método pode ser executada de varias formas (ou polimórficamente). Quem decide a **"forma"** é o objeto que recebe a chamada!

* Se um objeto a chama um método gritar() de um objeto b, então o objeto b decide a forma de implementação do método. Mais especificamente amada, é o tipo do objeto b que importa. Então a chamada do b.grita() vai ser um grito canino de b for Cão, felino se b for Gato e será equino se for um Cavalo.

O que importa, portanto, é o tipo de objeto receptor “b”. *

Nota do Autor

→ Onde ocorre o polimorfismo na linguagem Java?

R: chamadas de “métodos de objetos”.

→ Quais métodos?

R: Todos os métodos de objetos, não há polimorfismo ao chamar métodos estáticos (“métodos de classes”).

Não é qualquer objeto que pode gritar. Se eu tiver um objeto b representando uma cadeira e fizer b.gritar(), ..., cadeira não grita. Temos portanto que indicar o tipo de objeto que pode ser usado neste lugar

Classes e métodos abstratos

Em alguns casos é útil declarar classes para as quais o programador nunca pensará em instanciar objetos. Essas classes são chamadas classes abstratas. Elas somente são utilizadas como superclasses em hierarquias de herança, são chamadas superclasses abstratas. Não podemos ser utilizadas para instanciar por que são incompletas. As subclasses devem declarar as partes ausentes.

O propósito de uma classe abstrata é principalmente fornecer uma superclasse apropriada a partir da qual outras classes podem herdar e assim compartilhar um projeto comum.

Criação de uma classe abstrata: declare-a com abstract. Uma classe abstrata normalmente contém um ou mais métodos abstratos. Um método abstrato é um com a palavra-chave abstract.

```
Public abstract class nome{  
  
    Public abstract void metodoAbstrato();  
  
    Public void metodoComum(){  
        //...  
    }  
}
```

Exemplo:

```
public abstract class Animal {  
    protected String nome;  
    public abstract void gritar();  
    public void andar(){  
        System.out.println("Estou andando...");  
    }  
    public final void respirar(){  
        System.out.println("Respirando...");  
    }  
}
```

Clausula *instanceOf*

Verifica o tipo de instancia, ou seja, com que classe foi instanciado o objeto: **if (objeto instanceof Classe)...**

```
Ex: String x = "algo";  
X instanceof String = true  
X instanceof Pessoa = False
```

Downcasting

Uma declaração de superclasse só pode ser utilizada para invocar os métodos declarados na superclasse. Se o programador precisar realizar uma operação específica na subclasse em um objeto de subclasse referenciado por uma variável de superclasse, o programador deverá primeiro fazer uma coerção da referência de superclasse para uma referência de subclasse por meio de uma

técnica conhecida como downcasting. Isso permite ao programador invocar métodos de subclasse que não estão na superclasse.

Subclasse objeto = (Subclasse) objeto_superclasse;

Nota do Autor

Obs: atribuições entre variáveis de superclasse e subclasse.

- ➔ Atribuir referencia de **superclasse** a uma variável **de superclasse**
- ➔ Atribuir referencia de uma **subclasse** a uma variável **de subclasse**
- ➔ Atribuir referencia de **subclasse** a uma variável **de superclasse**

Atribuir referencia de **superclasse** a uma variável **de subclasse** só pode se a superclasse sofrer coerção explícita para o tipo de subclasse (downcasting).

Sombreamento

Sombreamento (*shadowing*) é a capacidade de poder definir duas, ou mais, variáveis com o mesmo nome em escopos diferentes. O código a seguir apresenta o exemplo clássico:

```
class UmaClasse {  
  
    String nome; // variável no escopo "classe"  
  
    public void setName ( String nome ){ // variável no escopo "método"  
        this.nome = nome;  
    }  
}
```

O sobreamento permite que o mesmo nome seja utilizado para duas variáveis diferentes. No caso, a variável "nome" definida na classe e a variável "nome" definida no método.

O detalhe com o uso de sobreamento é que as variáveis de maior escopo podem interagir com as de menor escopo. Contudo, como elas têm o mesmo nome, é necessário distingui-las. Para isso, é utilizada a palavra reservada `this` que representa o objeto corrente e contém, portanto, variáveis de escopo de classe. Caso o `this` não fosse utilizado junto de 'nome', o compilador

assume que você está se referindo à variável de menor escopo; no caso a definida no método. Isso não é uma falha. É a utilidade do sobreamento.

O compilador Java é um tanto esperto e avisa o programador de falhas básicas. Uma delas é a tentativa de atribuir uma variável a ela própria. Isso é um código que não tem nenhum propósito e o compilador o avisará quando detectar essa situação. Por isso se você escrever o código seguinte:

```
class UmaClasse {  
  
    String nome; // variável no escopo "classe"  
  
    public void setName ( String nome ) { // variável no escopo "método"  
        nome = nome;  
    }  
}
```

Sobrecarga

Sobrecarga (*overload*) é a capacidade de poder definir dois, ou mais métodos, numa mesma classe com o mesmo nome. Para que exista sobrecarga não é necessário que a linguagem seja orientada a objetos e, por isso, à semelhança do sobreamento a sobrecarga é normalmente entendida com uma característica da linguagem e não como uma forma de polimorfismo.

Embora os métodos possam ter o mesmo nome, eles têm obrigatoriamente que ter uma assinatura diferente. Eis alguns exemplos:

```
public abstract class Calculadora {  
  
    public abstract int calculaIdade ( int ano , int mes, int dia );  
    public abstract int calculaIdade ( Date data );  
    public abstract int calculaIdade ( Calendar data );  
}
```

Sobre-escrita

Sobre-escrita (*overriding*) é a capacidade de poder redefinir a implementação de um método que já foi definido e implementado em uma classe superior na hierarquia de herança.

Para que exista sobre-escrita é necessário que o método seja definido com a exata assinatura que existe na classe superior.

Exemplo: criamos uma classe Somador, com um método calculaSoma, ou seja, todo somador poderá calcular a soma de uma sequência de números fornecendo seu início até seu final.

```
public class Somador {  
    public int calculaSoma ( int inicio, int fim ){  
        int soma = 0 ;  
        for ( int i = inicio ; i <= fim ; i++ )  
            soma += i;  
        return soma;  
    }  
}
```

Digamos que a subclasse SomadorInteligente, tem outra maneira de somar essa sequência, ele pode sobrescrever esse método, ou seja, apagar a implementação antiga e criar uma nova para esse método.

```
public class SomadorInteligente extends Somador {  
    public int calculaSoma ( int inicio, int fim ){  
        int umAteInicio = inicio*( inicio+ 1 ) / 2 ;  
        int umAteFim = fim*( fim+ 1 ) / 2 ;  
        return umAteFim-umAteInicio;  
    }  
}
```

Interfaces

Representa operações comuns. Uma interface Java declara um conjunto de métodos. Um objeto que suporte uma interface deve possuir uma implementação para todo método declarado na interface. Uma interface é utilizada quando classes dispares (isto é, não relacionadas) precisam compartilhar métodos e constantes comuns. Isso permite que objetos de classes não relacionadas sejam processados a funcionalidade desejada e então implementar essa interface em qualquer classe que requerem essa funcionalidade.

Uma interface costuma ser utilizada no lugar de uma classe abstract quando não há nenhuma implementação padrão a herdar.

```
public interface Animal {  
    public void gritar();  
    public void andar();  
    public void respirar();  
}
```

Para utilizar uma interface, uma classe concreta deve especificar que implementa a interface e deve declarar nesta cada método com a assinatura especificada na sua declaração de interface. Uma classe que não implementa todos os métodos da interface é uma classe abstract e deve ser declarada abstract.

```
public class Cachorro implements Animal {  
    @Override  
    public void gritar(){  
        System.out.println("AuAuAu!!");  
    }  
    @Override  
    public void andar() {  
        // TODO Auto-generated method stub  
    }  
    @Override  
    public void respirar() {  
        // TODO Auto-generated method stub  
    }  
}
```

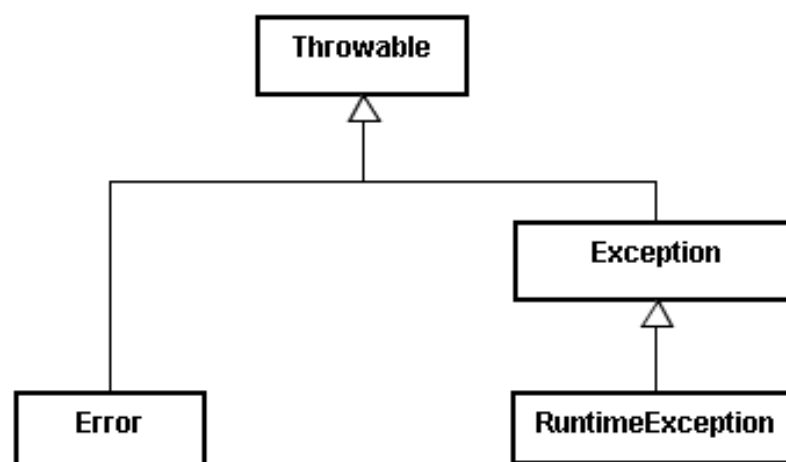
Erros e Exceções

Uma exceção é uma indicação de um problema que ocorre durante a execução de um programa. O nome 'exceção' dá a entender que o problema ocorre raramente. O tratamento de exceção permitem que os programas fiquem robustos e tolerantes a falhas, isto é, programas que sejam capazes de lidar com possíveis problemas e continuar executando. Algumas exceções:

- **ArrayIndexOutOfBoundsException:** ocorre quando uma tentativa de acessar um elemento é feita depois do fim de um array.
- **ClassCastException:** ocorre quando se tenta fazer coerção de um objeto que não tem um relacionamento 'é um' com o tipo especificado no operador de coerção.
- **NullPointerException:** ocorre sempre que uma referencia null é utilizada onde um objeto é esperado.(o objeto não existe ou não foi instanciado)
- **ArithmeticException:** ocorre quando divisão por zero dentre vários problemas da aritmética ocorrem..
- **InputMismatchException e NumberFormatException:** ocorre quando o método recebe uma String que não representa um valor válido.
- **NegativeArraySizeException:** ocorre quando o valor do índice que acessa o array é negativo (não existe posição negativo em array).

Tipos de Exceção

Existem três categorias de exceções: Erro, Falha e Exceção de Contingência representadas respectivamente pelas classes: Error, RuntimeException e Exception. Todas estas classes são filhas de Throwable. A hierarquia de exceções em Java não tem como objetivo criar implementações ligeiramente diferentes da mesma coisa. Cada tipo de exceção tem uma interpretação especial que se reflete na forma como o programador tem que lidar com elas.



Erros

Erros são exceções tão graves que a aplicação não tem o que fazer. São erros todas as classes que descendem diretamente de `Error`.

É importante que os erros sejam reportados e que se saiba que aconteceram, mas o programa não tem o que fazer para resolver o problema que eles apontam. Erros indicam que alguma coisa está realmente muito errada na construção do código ou no ambiente de execução. Exemplos de erros são `OutOfMemoryError` que é lançada quando o programa precisa de mais memória mas ela não está disponível, e `StackOverflowError` que acontece quando a pilha estoura, por exemplo, quando um método se chama a si mesmo sem nunca retornar.

Falhas

Falhas são exceções que a aplicação causa e pode tratar. Digo pode tratar porque não é obrigada a fazê-lo. São falhas todas as classes que descendem diretamente de `RuntimeException`.

Se a aplicação nunca apanhar este tipo de exceção, tudo bem, a JVM irá capturá-la. Mas provavelmente a sua aplicação não mais funcionará corretamente. Exemplos de falhas são `IllegalArgumentException` e `NullPointerException`.

A primeira acontece quando se passa um parâmetro para um método e o método não o pode usar. A segunda acontece sempre que tentar invocar um método em uma variável de objeto não inicializada. Isso é bastante comum e por isso ela é, provavelmente, a exceção mais reportada de todas. Exceções deste tipo existem em outras linguagem que têm o conceito de exceção e são as que nos devem preocupar enquanto programamos porque traduzem situações que desafiam a lógica do programa.

Exceções de Contingência

Exceções de Contingência são aquelas que a aplicação pode causar ou não, mas que tem que tratar explicitamente. Exceções de Contingência são todas aquelas que descendem diretamente de `Exception` excepto as que descendem de `RuntimeException`.

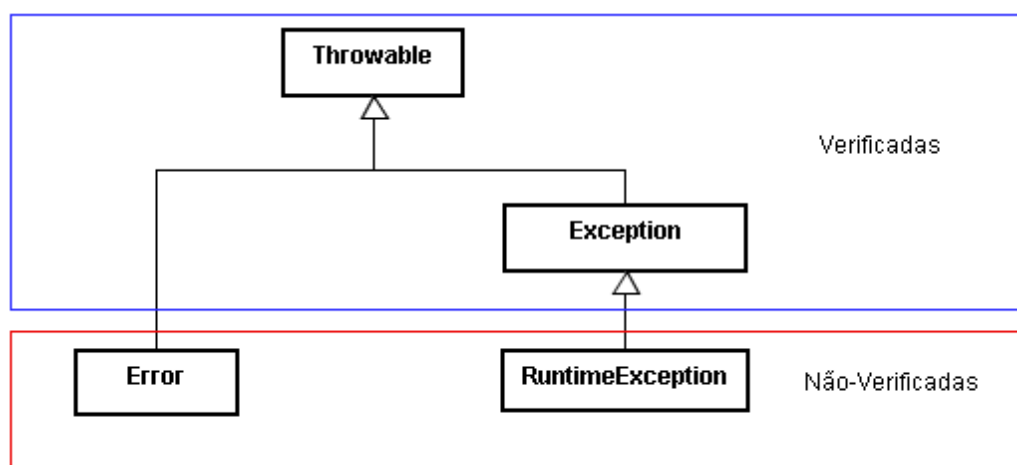
Exceções verificadas e não-verificadas

Java foi a primeira linguagem a introduzir o conceito de exceção verificada. Por padrão as exceções em Java são verificadas.

A aplicação é obrigada a tratar estas exceções explicitamente na medida que o método que recebe este tipo de exceção é forçado a verificar se pode resolver o problema. Se o método sabe que não poderá resolver a exceção, ele deve declarar isso explicitamente.

Note que este mecanismo de verificação é muito útil quando temos que antever planos de contingência para a ocorrência da exceção. O uso da palavra contingência não é coincidência. Java entende que para todas as exceções devem existir planos de contingência, alternativas que permitam que o programa continue funcionando normalmente. Assim todas as classes que descendem diretamente de Throwable ou Exception são exceções verificadas.

Mas o mundo não é perfeito e Java entende também que existem exceções para as quais não é possível antever um plano de contingência, erros e falhas, são esses tipos. Os primeiros nunca deveriam acontecer, portanto não faz sentido ter planos para os resolver. Os segundos podem até demonstrar que a aplicação está funcionando corretamente ao identificar aquele problema. Por isso Error e RuntimeException são exceções não-verificadas.



O fato de RuntimeException herdar de Exception confunde muita gente porque parece significar que todas as RuntimeException são também Exception. Isso significaria que todas as falhas poderiam ser resolvidas se existirem planos de contingência para elas. O que é verdade todas as falhas são possíveis exceções de contingência. Na verdade é por isso que todas as RuntimeException são também Exception. Elas podem ser resolvidas se o programa tiver como e desse ponto de vista a sua resolução é igual à de Exception.

Mas RuntimeException herdar de Exception parece significar que o comportamento de exceção verificada é também herdado, o que não é verdade.

O mesmo argumento poderia ser usado com `Error` e `Throwable`. A verificação é algo que o compilador obriga, como obriga por exemplo que não exista nenhum código depois de um `return` o que seja feito um `cast` se os dois lados de uma atribuição não são da mesma classe. A verificação é portanto uma característica da linguagem que nada tem a ver com herança. Se este conceito é difícil de entender, pense apenas que as várias classes de exceção são marcadores para o compilador e a JVM saberem como e quando lançar/capturar a exceção que elas representam. Exceções verificadas são uma característica da inteligência da linguagem Java.

Tratamento de Exceções

```
try{
    //código que podem lançar exceção
    //se ocorrer uma exceção o restante do código
    //é pulado
}catch(Exception e){
    //códigos a serem executados quando a exceção for lançada
    //pelo menos um bloco catch deve seguir o bloco try
}finally{
    //instruções a serem executadas mesmo que
    //alguma exceção foi lançada.
    //ou seja, de qualquer jeito, deve-se executar
    //essas instruções
}
```

Cada bloco **catch** pode ter apenas um parâmetro e tratar apenas uma exceção. Se ocorrer uma exceção em um bloco **try**, o bloco termina imediatamente e o controle do programa é transferido para o primeiro dos blocos **catch** seguintes em que o tipo do parâmetro de exceção corresponde ao tipo de exceção lançada. Após a exceção ser tratada, o controle do programa não retorna ao ponto de lançamento porque o bloco `try` expirou. Exemplo que força exceções:

```
try{

    //NegativeArraySizeException
    String[] teste = new String[-5];
    //ArrayIndexOutOfBoundsException
    teste[5]="";

    //NullPointerException
    String s = null;
```

```
s.toString();

//NumberFormatException
int i = Integer.parseInt("oi");

}catch(NegativeArraySizeException n){
    System.out.println("Foi lançada: "+n);
}catch(ArrayIndexOutOfBoundsException n){
    System.out.println("Foi lançada: "+n);
}catch(NullPointerException n){
    System.out.println("Foi lançada: "+n);
}catch(NumberFormatException n){
    System.out.println("Foi lançada: "+n);
}
```

Criando Exceções

A maioria dos programadores em Java utiliza as classes existentes da API do Java, fornecedores independentes e bibliotecas de classe livremente disponíveis (normalmente descarregáveis a partir da Internet) para construir aplicativos Java. Em geral, os métodos dessas classes são declarados para lançar exceções apropriadas se ocorrer algum problema. Os programadores escrevem um código que processa essas exceções existentes para tornar os programas mais robustos. Se for um programador que constrói classes que serão utilizadas por outros programadores, você pode achar útil declarar suas próprias classes de exceção que são específicas aos problemas que podem ocorrer quando outro programadorempregar suas classesreutilizáveis.

Uma nova classe de exceção deve estender uma classe de exceção existente para assegurar que a classe pode ser utilizada com o mecanismo de tratamento de exceções. Como qualquer outra classe, uma classe de exceção pode conter campos e métodos. Entretanto, uma nova classe de exceção típica contém somente dois construtores - um que não aceita nenhum argumento e passa uma mensagem de exceção padrão para o construtor da superclasse, e um que recebe uma mensagem de exceção personalizada como uma String e a passa para o construtor da superclasse. Por convenção, todos os nomes de classe de exceções devem terminar com a palavra Exception.

Para lançar uma exceção simplesmente usamos a clausula throw seguida do objeto que representa a exceção que queremos lançar. O conceito é semelhante ao de return, mas enquanto return está devolvendo um resultado de dentro do método, throw está lançando uma

exceção. Nunca é possível considerar uma exceção como o resultado de um método, o objetivo do método é obter resultados sem lançar exceções.

Lembre-se que exceções só podem ser lançadas de dentro de métodos e que uma vez lançadas elas passam por toda a cadeia de métodos que estão na pilha de chamadas. Ou seja, passam pelo método que chamou o método que lançou a exceção. Pelo método que chamou esse método, e pelo método que chamou este outro e assim sucessivamente até que sejam apanhadas. Se o programa não apanhar a exceção a JVM o fará. Por padrão, a JVM exibirá uma mensagem no console.

Exemplo: Queremos verificar se o Sexo de uma pessoa foi digitado corretamente e sua idade não é negativa ou nula. Criamos duas exceptions para ser lançadas. Uma extends a Exception e outra a RuntimeException. Qual é a diferença? A RuntimeException, não precisa ser tratada caso o programador não queira. Já a Exception, é obrigatório. O uso das duas aqui, mostra apenas a diferença, fica a seu gosto e gosto das regras de negócios do seu projeto de sistema.

```
public class IdadeNegativaException extends Exception{

    public IdadeNegativaException(){
        super("IdadeNegativaException: Idade não pode ser negativa!");
    }

}

public class SexoInvalidoException extends RuntimeException {

    public SexoInvalidoException(){
        super("SexoInvalidoException: apenas M ou F");
    }

}

public class Pessoa {

    public String nome;
    public int idade;
    public String sexo;

    public String getNome() {}
    public void setNome(String nome) {}
    public int getIdade() {}
}
```

```
public void setIdade(int idade) throws IdadeNegativaException{
    if( idade<=0 ) throw new IdadeNegativaException();
    this.idade = idade;
}

public String getSexo() {

public void setSexo(String sexo) throws SexoInvalidoException {
    if( "M".equalsIgnoreCase(sexo) || "F".equalsIgnoreCase(sexo)){
        this.sexo = sexo;
    }else{
        throw new SexoInvalidoException();
    }
}

}
```

Repare que, utilizei os métodos setSexo e setIdade para fazer a verificação. Aqui temos uma visão mais clara da importância do encapsulamento. Deve-se declarar que ambos os métodos lançam uma exceção, para isso usamos a cláusula throws e indicamos qual exceção, nota: podem ser lançadas quantas exceções o programador quiser, basta que ele indique-as separando por vírgula. Nos pontos onde podem ser lançadas as exceções, usamos a cláusula throw new e o construtor da exceção. Agora vamos testar:

```
public class TestePessoa {

    public static void main(String[] args) {

        Pessoa p = new Pessoa();

        //não vai lançar exceção, porém vai dar um erro,
        //pois essa exceção deve ser tratada
        p.setIdade(14);

        //agora que circundamos com try/catch, não haverá problemas
        try {
            p.setIdade(-6);
        } catch (IdadeNegativaException e) {
            e.printStackTrace();
        }

        //não vai lançar exceção, e não é preciso tratar,
        //já que ele é uma RuntimeException, e seu tratamento
        //não é obrigatório
        p.setSexo("m");
    }
}
```



```
//agora que circundamos com try/catch
try {
    p.setSexo("s");
} catch (SexoInvalidoException e) {
    e.printStackTrace();
}

}
```

Como resultado temos:

```
exe02 Criando Excecoes.IdadeNegativaException: IdadeNegativaException: Idade não pode ser negativa!
at exe02_Criando_Excecoes.Pessoa.setIdade(Pessoa.java:20)
at exe02_Criando_Excecoes.TestePessoa.main(TestePessoa.java:16)
exe02 Criando Excecoes.SexoInvalidoException: SexoInvalidoException: apenas M ou F
at exe02_Criando_Excecoes.Pessoa.setSexo(Pessoa.java:32)
at exe02_Criando_Excecoes.TestePessoa.main(TestePessoa.java:30)
```

Try-Finally

Por vezes, mesmo sabendo que os métodos que estamos usando lançam exceções, sabemos também que não podemos fazer nada para as resolver. Nesse caso, simplesmente não usamos o bloco try-catch e simplesmente declaramos as exceções com throws na assinatura do método. Mas, e se, mesmo acontecendo uma exceção existe um código que precisamos executar? É neste caso que usamos o bloco finally.

Este tipo de problema é mais comum do que possa parecer. Por exemplo, se você está escrevendo num arquivo e acontece um erro, o arquivo tem que ser fechado mesmo assim. Ou se você está usando uma conexão a banco de dados e acontece algum problema a conexão tem que ser fechada.

Para usar o bloco try-finally, começamos como envolver os métodos que podem lançar exceções como vimos antes, mas usamos um bloco finally em vez de um catch.

```
try { // aqui executamos um método que pode lançar uma exceção que não //
sabemos resolver} finally { // aqui executamos código que tem que ser executado,
mesmo que um problema aconteça.}
```

Isto é muito útil, mas pense o que acontece se dentro do bloco try colocamos um return.

Isso significa que algo tem que ser retornado para fora do método, mas significa também que o método acaba aí. Nenhum código pode ser executado depois de um return (o compilador vai-se queixar dizendo que o código seguinte

é inalcançável). Isso é tudo verdade, exceto se esse código suplementar estiver dentro de um bloco finally. O código dentro do bloco finally não apenas é executado se uma exceção acontecer, mas também se o método for interrompido. É garantido que o código dentro do bloco finally sempre será executado, aconteça o que acontecer. Este é um outro uso importante deste bloco.

Genéricos

Fornecem um meio de criar modelos gerais. Métodos genéricos ou classes genéricas permitem que programadores especifiquem, com uma única declaração de método, um conjunto de métodos relacionados ou, com uma única declaração de classe, um conjunto de tipos relacionados, respectivamente.

Métodos e classes genéricas estão entre as capacidades mais poderosas do Java para reutilização de software com segurança de tipo em tempo de compilação.

Métodos sobrecarregados são frequentemente utilizados para realizar operações semelhantes em tipos diferentes de dados. Exemplo:

```
public class Genericos {  
    public static void printArray(Integer[] inputArray){  
        for(Integer e: inputArray)  
            System.out.println(e);  
    }  
    public static void printArray(Double[] inputArray){  
        for(Double e: inputArray)  
            System.out.println(e);  
    }  
    public static void printArray(Character[] inputArray){  
        for(Character e: inputArray)  
            System.out.println(e);  
    }  
}
```

Se as operações realizadas por vários métodos sobrecarregados forem idênticas para cada tipo de argumento, os métodos sobrecarregados podem ser codificados mais compacta e convenientemente com um método genérico.

Pode-se escrever uma única declaração de método genérico que pode ser chamada com argumentos de tipos diferentes.

Se substituirmos os tipos dos elementos em cada método por um nome genérico então todos os três métodos seriam semelhantes a:

```
public static <E> void printArray (E[] inputArray){  
    for (E e : inputArray)  
        System.out.println(e);  
}
```

Todas as declarações de métodos genéricos têm uma seção de parâmetros de tipo delimitada por <> que precedem o tipo de retorno do método.

Cada seção de parâmetros de tipo contem um ou mais parâmetros de tipos. Um parâmetro de tipo é um identificador que especifica um nome genérico do tipo. Os parâmetros de tipo podem ser utilizados para declarar o tipo de retorno, tipos de parâmetros e tipos de variáveis locais em uma declaração de método genérico e atuam como marcadores de lugar para os tipos de argumentos passados ao método genérico, conhecidos como argumentos de tipos reais.

Classes genéricas – um exemplo

O conceito de uma estrutura de dados, como uma pilha, pode ser entendido independente do tipo de elemento que ela manipula. Classes genéricas fornecem um meio de descrever o conceito de uma pilha de uma maneira independente do tipo.

Uma classe Stack(Pilha) genérica, poderia ser a base para criar muitas classes Stack (Stack de Double, Stack de Integer,...). essas classes são conhecidas como classes parametrizadas ou tipos parametrizados porque aceitam um ou mais parâmetros.

```
public class Stack <E>{  
    private E[] elements;  
  
    public Stack(int s){  
        //...  
        elements = (E[]) new Object[s];  
    }  
  
    //...  
    public E pop(){  
        E e = (E) new Object();  
        //...  
        return e;  
    }  
}
```

Ao criar um objeto dessa classe:

```
public class TesteStack {  
  
    private Stack<Integer> intPilha = new Stack<Integer>(5);  
    private Stack<Double> doublePilha = new Stack<Double>(10);  
    private Stack<String> stringPilha = new Stack<String>(5);  
  
}
```

Tipos brutos

Também é possível instanciar uma classe genérica Stack sem especificar um argumento de tipo, como a seguir:

```
private Stack objectPilha = new Stack(5);
```

O compilador utiliza implicitamente o tipo object por toda a classe genérica para cada argumento de tipo.

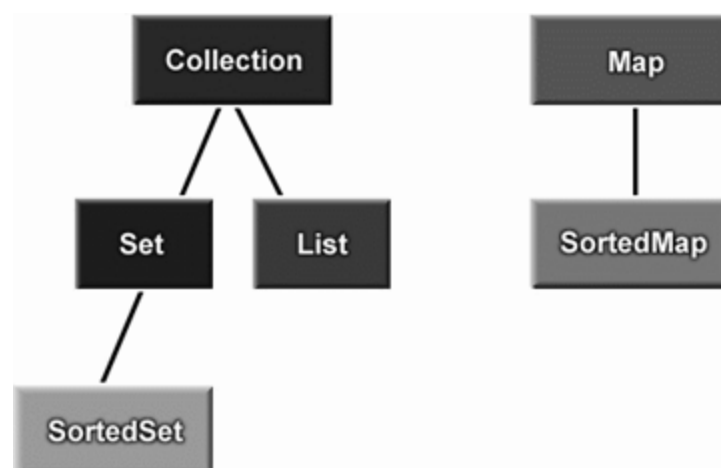
Coleções

Com as coleções, os programadores utilizam estruturas de dados existentes, sem se preocupar com a maneira como elas são implementadas.

Uma coleção é uma estrutura de dados que pode armazenar referências a outros objetos. As interfaces de estrutura de coleção declaram as operações a ser realizadas genericamente em vários tipos de coleções.

A estrutura de coleções fornece implementações de alto desempenho e alta qualidade de estruturas de dados comuns e permite a reutilização de software.

Esses recursos minimizam a quantidade de codificação que os programadores devem fazer para criar e manipular coleções. Você pode especificar o tipo exato que será armazenado em uma coleção.



- **Collection** -> interface-raiz na hierarquia de coleções, está no topo da hierarquia definindo operações que são comuns a todas as coleções.
- **Set** -> está relacionada a ideia de conjuntos e define uma coleção que não contem duplicatas.
- **List**-> é uma coleção ordenada, que ao contrario da interface Set, pode conter duplicatas. Além disso, temos controle total sobre a posição onde se encontra cada elemento de nossa coleção, podendo acessar cada um deles pelo índice.
- **Queue** -> representa uma fila

- **Map** -> associa chaves a valores e não pode conter valores duplicados.

Listas

Uma lista é uma collection ordenada que pode conter elementos duplicados. A interface List é implementada por várias classes, incluídas as classes ArrayList, LinkedList e Vector.

A classe ArrayList e classe vector são implementações de arrays redimensionáveis de List.

A classe LinkedList é uma implementação de Lista encadeada da list. LinkedList podem ser utilizadas para criar pilhas, filas, arvores e dequeues.

Exemplo de criação de listas:

```
//declaração sem generics
List listaQualquer = new LinkedList();

//declaração com generics
List<String> nomes = new LinkedList<String>();
```

Nota do Autor

Obs: vantagens de usar Generics nas Collections:

- Restringe os tipos de dados usados pela coleção, facilitando a consulta e manipulação dos objetos armazenados na coleção.
- facilita, pois dispensa o uso de casting e dispensa a necessidade de se conhecer o tipo do objeto armazenado na corrente iteração.

O uso de Generics significa que estamos utilizando/especificando um tipo único do objeto para toda a coleção.

ArrayList

Construção:

ArrauList(); -> normal

ArrayList(Collection c); -> constrói a lista contendo os elementos de c.

ArrayList(int i); -> constrói com uma capacidade delimitada.

Alguns métodos:

Add: adiciona elemento especificado, pode-se especificar também o índice `add(elemento)` ou `add(índice, elemento)`;

Clear: remove todos os elementos da lista;

Get(index): retorna o elemento da posição indicada;

isEmpty(): retorna true se estiver vazio;

remove(index): remove o elemento da posição indicada;

removeRange(index): remove os elementos cujo índice está entre início e fim;

set(index, elemento): substitui o elemento do index por outro;

size: retorna o número de elementos nesta lista

LinkedList

Usando:

```
List<Tipo> listanome = new LinkedList<Tipo>();
```

Além dos mesmos métodos de `ArrayList` descritos, `LinkedList` implementa:

addFirst(), addLast(): insere o elemento no início e no fim da lista, respectivamente.

getFirst(), getLast(): retorna o elemento no início e no fim da lista, respectivamente.

removeFirst(), removeLast(): remove o elemento no início e no fim da lista, respectivamente.

Algoritmos de coleção

São algoritmos com algumas funções já prontas para se usar com coleções. Como usar:

```
Collections.<método>(<coleção>);
```

Collections.sort(lista) -> classifica os elementos de uma lista, que deve implementar `Comparable`. A ordem é determinada pelo ordenamento natural do tipo dos elementos como implementado por sua classe pelo método **compareTo**.

Collections.shuffle(lista) ->o algoritmo shuffle ordena aleatoriamente elementos de uma coleção.

Collections.reverse(lista) -> inverte a ordem dos elementos da coleção.

Collections.fill(lista,elemento) ->sobrescreve os elementos da coleção com um valor especificado;

Collections.copy(copyLista,lista) -> recebe dois argumentos (uma coleção de destino e uma coleção de origem). Cada lemento da coleção de origem é copiado para a coleção de destino.

Collections.min(list) e **Collections.max(list)** ->retornam, min (o menor) e max (o maior) valor em uma coleção.

Collections.binarySearch(list, key) ->o algoritmo binarySearch localiza um objeto em uma coleção. Se o objeto for encontrado, seu índice é retornado. Se o objeto não for localizado, binarySearch retorna um valor negativo. **O método espera que os elementos da lista estejam em ordem crescente.**

Collections.addAll(list, array)-> Aceita dois argumentos (uma coleção na qual inserir os novos valores e um array que fornece elementos a ser inserido).

Collections.frequency(list, objeto)->aceita dois argumentos (um collection a ser pesquisado e um objeto a ser procurado na coleção). Retorna o numero de vezes que o segundo argumento aparece na coleção.

Collections.disjoin(lista1, lista2)->aceita dois collections e retorna true se não tiverem nenhum elemento em comum.

Stack(Pilha)

```
Stack<String> pilha = new Stack<String>(); //cria a pilha
pilha.push("objeto"); //insere um objeto na pilha
String objeto = pilha.pop(); // remove um objeto na pilha
```


Queue(Fila)

```
PriorityQueue<String> queue = new PriorityQueue<String>(); //cria a fila
queue.offer("objeto"); //insere na fila
objeto = queue.peek(); //retorna o 1º da fila
queue.poll(); //remove o 1º da fila
```

Conjuntos

Um set é uma Collection que contem elementos únicos (isto é, elementos não duplicados). A estrutura de coleções contem diversas implementações de Set, incluindo **HashSet** (armazena seus elementos em uma tabela de hash) e **TreeSet** (armazena seus elementos em uma árvore).

```
Set<String> set = new HashSet<String>(nomes);
```

A estrutura de coleções também inclui a interface SortedSet (que estende Set) para conjuntos que mantem seus elementos na ordem de classificação. A classe TreeSet implementa SortedSet.

```
SortedSet<String> ss = new TreeSet<String>();
ss.headSet("objeto"); //retorna os elementos anteriores ao objeto
ss.tailSet("objeto"); //retorna os elementos posteriores ao objeto
ss.first(); //retorna o primeiro elemento do conjunto
ss.last(); //retorna o ultimo elemento do conjunto
```

Mapas

São chamados de array associativos. Maps associam chaves aos valores e não podem conter chaves com duplicidade (cada chave pode mapear somente um valor, isso é chamado mapeamento um para um). O conteúdo de um map pode ser visto e manipulado como coleção.

```
Map<Integer,String> mapa = new HashMap<Integer,String>();
mapa.put(1,"objeto e sua chave"); //insere um objeto com sua chave no mapa
mapa.entrySet(); //retorna um conjunto (Set) com todos os elementos (chave-valor)
mapa.keySet(); //retorna uma coleção (collection) com todas as chaves
mapa.values(); //retorna uma coleção (collection) com todos os valores
mapa.containsKey(1); //verifica se a chave está no mapa
```

Exercícios resolvidos

1. Faça um programa em Java que cria um vetor de inteiros com tamanho definido pelo usuário em tempo de execução do programa.

Preencha o vetor com valores aleatórios gerados pelo computador e encontre o menor elemento que foi armazenado no vetor exibindo o elemento e seu respectivo índice onde ele se encontra no vetor.

```
import java.util.Random;

public class Menor {

    public int[] vetor; //vetor de numeros
    public int tam; //indica o tamanho do vetor

    public static void main(String[] args) {

        Menor m = new Menor();
        m.criarVetor();
        m.preencher();
        m.mostrarVetor();
        System.out.println("O menor valor é:" + m.vetor[ m.getIndexMenor() ]);

    }

    public void criarVetor(){
        int tam; //indica o tamanho do vetor
        do{
            tam = Integer.parseInt(
                JOptionPane.showInputDialog(null,
                    "Entre com um número maior que zero e inteiro.",
                    "Tamanho do vetor de Números",
                    JOptionPane.QUESTION_MESSAGE));

            if (tam<=0) continue; //verifica se o valor é negativo ou zero,
                                //se for pula as próximas ações

            try{
                vetor = new int[tam]; //cria o vetor
                break; //dando certo, termina o laço
            }catch(NumberFormatException e){
                continue;
            } // caso seja lançada, o laço continua;

        }while(true); //o laço irá se repetir enquanto o valor digitado for real,
                    //negativo ou zero.
```

```

    }

    public void preencher(){
        Random r = new Random();
        for(int i=0; i<vetor.length; i++)
            vetor[i]= r.nextInt(10);
    }

    public int getIndexMenor(){
        if(vetor.length==1) return 0; //se o vetor tiver apenas um numero,
                                     //retorna ele mesmo
        int indexMenor =0; //assumimos que o menor valor é o primeiro

        for(int i=0; i<vetor.length; i++){
            if(vetor[indexMenor] > vetor[i]) indexMenor=i;
        }

        return indexMenor;
    }

    public void mostrarVetor(){
        System.out.print("O vetor:");
        for(int i : vetor){
            System.out.print(i+" ");
        }
        System.out.println("");
    }
}

```

2. Crie uma hierarquia: Pessoa, Aluno, Professor. Aluno deverá conter 4 notas, ra, atividade, curso, media. Professor deverá conter salario e disciplina. Pessoa deverá conter nome, idade e telefone.

Crie uma classe que implemente isso, utilizando um vetor de aluno e outro de professores. O programa deverá mostrar todos os dados de Aluno e Professores. Ao final mostre os alunos aprovados, cujas médias maiores que a média da sala.

```

public class Pessoa {

    private String nome;
    private int idade;
    private int telefone;
}

```

```
public Pessoa() {
    super();
}

public Pessoa(String nome, int idade, int telefone) {
    super();
    this.nome = nome;
    this.idade = idade;
    this.telefone = telefone;
}

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public int getIdade() {
    return idade;
}

public void setIdade(int idade) {
    this.idade = idade;
}

public int getTelefone() {
    return telefone;
}

public void setTelefone(int telefone) {
    this.telefone = telefone;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
```

```
        result = prime * result + ((nome == null) ? 0 :
            nome.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Pessoa other = (Pessoa) obj;
        if (nome == null) {
            if (other.nome != null)
                return false;
        } else if (!nome.equals(other.nome))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return nome;
    }

}

public class Aluno extends Pessoa {

    private int ra;
    private Float[] notas;
    private Float atividade;
    private double media;
    private String curso;

    public Aluno() {
        super();
        notas = new Float[4];
    }
}
```

```
public Aluno(int ra, Float[] notas, Float atividade,
             double media,String curso) {
    super();
    this.ra = ra;
    this.notas = notas;
    this.atividade = atividade;
    this.media = media;
    this.curso = curso;
}

public int getRa() {
    return ra;
}

public void setRa(int ra) {
    this.ra = ra;
}

public Float[] getNotas() {
    return notas;
}

public void setNotas(Float[] notas) {
    this.notas = notas;
}

public Float getAtividade() {
    return atividade;
}

public void setAtividade(Float atividade) {
    this.atividade = atividade;
}

public double getMedia() {
    return media;
}

public void setMedia(double media) {
    this.media = media;
}
```

```
    public String getCurso() {
        return curso;
    }

    public void setCurso(String curso) {
        this.curso = curso;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = super.hashCode();
        result = prime * result + ra;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!super.equals(obj))
            return false;
        if (getClass() != obj.getClass())
            return false;
        Aluno other = (Aluno) obj;
        if (ra != other.ra)
            return false;
        return true;
    }

}

public class Professor extends Pessoa {

    private String disciplina;
    private Float salario;
```

```
    public Professor() {
        super();
    }

    public Professor(String disciplina, Float salario) {
        super();
        this.disciplina = disciplina;
        this.salario = salario;
    }

    public String getDisciplina() {
        return disciplina;
    }

    public void setDisciplina(String disciplina) {
        this.disciplina = disciplina;
    }

    public Float getSalario() {
        return salario;
    }

    public void setSalario(Float salario) {
        this.salario = salario;
    }
}

public class Professor extends Pessoa {

    private String disciplina;
    private Float salario;

    public Professor() {
        super();
    }

    public Professor(String disciplina, Float salario) {
        super();
        this.disciplina = disciplina;
        this.salario = salario;
    }
}
```



```
    public String getDisciplina() {  
        return disciplina;  
    }  
  
    public void setDisciplina(String disciplina) {  
        this.disciplina = disciplina;  
    }  
  
    public Float getSalario() {  
        return salario;  
    }  
  
    public void setSalario(Float salario) {  
        this.salario = salario;  
    }  
  
}
```

A classe teste:

```
public class Teste {  
  
    private Aluno[] alunos;  
    private Aluno a;  
    private float mediaSala;  
    private Professor[] professores;  
    private Professor p;  
  
    public static void main(String args[]){  
  
        Teste t = new Teste();  
  
        //alunos  
        t.iniciarAlunos(3);  
        t.mostrarAlunos();  
        //professores  
        t.iniciarProfessores(2);  
        t.mostrarProfessores();  
  
    }  
  
    //métodos de manipulação de alunos  
    public void iniciarAlunos(int qtde){..}
```

```
public void mostrarAlunos(){  
  
    //metodos de manipulação de professor  
    public void iniciarProfessores(int qtde){  
  
    public void mostrarProfessores(){
```

Métodos:

```
public void iniciarAlunos(int qtde){  
    Scanner sc = new Scanner(System.in);  
  
    //cria o vetor com uma certa quantidade de alunos  
    alunos = new Aluno[qtde];  
    mediaSala = 0; //para o calculo da média da sala  
  
    for(int i=0; i< alunos.length; i++){  
        try{  
  
            System.out.println("-----Dados do aluno "+(i+1)+"-----");  
            a = new Aluno();  
            //receber os dados do aluno  
            System.out.println("Entre com o Nome do aluno:");  
            a.setNome(sc.next());  
            System.out.println("Entre com o Curso do aluno:");  
            a.setCurso(sc.next());  
            System.out.println("Entre com o RA do aluno:");  
            a.setRa(sc.nextInt());  
            System.out.println("Entre com o Idade do aluno:");  
            a.setIdade(sc.nextInt());  
            System.out.println("Entre com o Telefone do aluno:");  
            a.setTelefone(sc.nextInt());  
  
            float soma=0; //para calcular a soma das notas do aluno  
            //receber as notas do aluno e calcular a soma delas  
            for(int y=0; y<4; y++){  
                System.out.println("Entre com o "+(y+1)+"ª nota do aluno:");  
                (a.getNotas())[i] = sc.nextFloat();  
                soma += (a.getNotas())[i];  
            }  
  
            System.out.println("Entre com o Atividade do aluno:");  
            a.setAtividade(sc.nextFloat());  
  
            //calcular a media  
            a.setMedia(soma/4);  
            //armazenar o aluno no vetor  
            alunos[i]=a;
```

```
//soma para calculo da media da sala
mediaSala += a.getMedia();

} catch (InputMismatchException e) {
    System.out.println("Entrada Inválida:");
    i--;
}
mediaSala /= alunos.length;
}

}

public void mostrarAlunos() {
    for (Aluno a : alunos) {
        //garante que apenas os alunos acima da media sejam mostrados
        if (a.getMedia() < mediaSala) continue;
        System.out.println("Aluno : " + a.getNome() + ", média: " + a.getMedia());
    }
}

public void iniciarProfessores(int qtde) {
    Scanner sc = new Scanner(System.in);

    //cria o vetor com uma certa quantidade de alunos
    professores = new Professor[qtde];

    mediaSala = 0; //para o calculo da média da sala

    for (int i = 0; i < professores.length; i++) {
        try {

            System.out.println("-----Dados do professor " + (i + 1) + "-----");
            p = new Professor();
            //receber os dados do aluno
            System.out.println("Entre com o Nome do professor:");
            p.setNome(sc.next());
            System.out.println("Entre com o Disciplina do aluno:");
            p.setDisciplina(sc.next());
            System.out.println("Entre com o Salario do aluno:");
            p.setSalario(sc.nextFloat());
            System.out.println("Entre com o Idade do Professor:");
            p.setIdade(sc.nextInt());
            System.out.println("Entre com o Telefone do aluno:");
            p.setTelefone(sc.nextInt());
            professores[i] = p;
        } catch (InputMismatchException e) {
            System.out.println("Entrada Inválida:");
            i--;
        }
    }
}

public void mostrarProfessores() {
    for (Professor p : professores) {
        System.out.println("Professor : " + p.getNome() + ", disciplina: " + p.getDisciplina());
    }
}
```

3. Implemente o seguinte polimorfismo:

a) Crie a interface Figura geométrica que deve conter o método:

public double calcularArea();

b) Crie a classe Quadrado que deve implementar a interface figura Geometrica e sobrescrever o método calcularArea(). Esta classe deve contar o atributo tamanholado.

calcularArea -> tamanhaLado * tamanholado

c) Crie a classe Triangulo que deve implementar a interface figuraGeometrica e sobrescrever o método calcularArea(). Esta classe deve conter os atributos base e altura.

calcularArea -> (base*altura)/2

d) Crie a classe Circulo que deve implementar a interface FiguraGeometrica e sobrescrever o método calcularArea(). Esta classe deve conter o atributo raio.

calcularArea -> Math.Pi*raio*raio

e) Crie a classe UsaFiguraGeometrica que deve conter um vetor contendo 3 figuras Geometricas:

- Instancie o vetor de FiguraGeometrica.

- Crie as FigurasGoemetricas em cada posição do vetor.

Posição 0 – Circulo

Posição 1 – Quadrado

Posição 2 – Triangulo

- Liste o vetor e informe a cada posição qual a figura geométrica (circulo, quadrado ou triangulo) e qual é a sua área.

```
public interface FiguraGeometrica {  
  
    public double calcularArea();  
    public String getInformacao();  
  
}
```

```
public class Quadrado implements FiguraGeometrica {

    public double lado;

    @Override
    public double calcularArea() {
        return lado*lado;
    }

    @Override
    public String getInformacao() {
        return "Quadrado, "+calcularArea()+"cm²";
    }

}

public class Circulo implements FiguraGeometrica {

    public double raio;

    @Override
    public double calcularArea() {
        return Math.PI*raio*raio;
    }

    @Override
    public String getInformacao() {
        return "Circulo";
    }

}

public class Triangulo implements FiguraGeometrica {

    public double base,altura;

    @Override
    public double calcularArea() {
        return (base * altura)/2;
    }

    @Override
    public String getInformacao() {
        return "Triangulo";
    }

}
```

```
public class UsaFiguraGeometrica {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        //criando o vetor de figuras.
        //repare que no mesmo vetor eu posso ter três tipos
        //diferentes, isso porque todos esse tres tipos
        //implementam a interface FiguraGeometrica.
        //ou seja, eles são uma FiguraGeometrica
        FiguraGeometrica fig[] = new FiguraGeometrica[3];
        fig[0] = new Circulo();
        fig[1] = new Quadrado();
        fig[2] = new Triangulo();

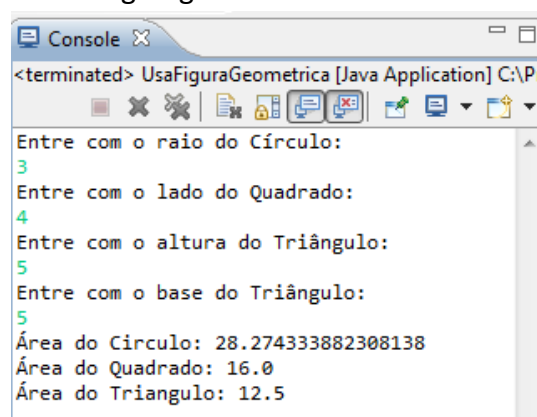
        //a seguir é realizado um cast no vetor, pois mesmo que cada indice
        //do vetor seja um tipo 'diferente', todos são tratados como FiguraGeometrica
        //assim, é preciso o cast para poder acessar os atributos específicos de cada um
        System.out.println("Entre com o raio do Círculo:");
        ((Circulo) fig[0]).raio = sc.nextDouble();

        System.out.println("Entre com o lado do Quadrado:");
        ((Quadrado) fig[1]).lado = sc.nextDouble();

        System.out.println("Entre com o altura do Triângulo:");
        ((Triangulo) fig[2]).altura = sc.nextDouble();
        System.out.println("Entre com o base do Triângulo:");
        ((Triangulo) fig[2]).base = sc.nextDouble();

        //seguir mostraremos as areas e de que figura é.
        //repare que para isso não é preciso realizar o cast
        //pois TODA figurageometrica tem os metodos calculaArea e getInformacao
        for(FiguraGeometrica fg : fig){
            System.out.println("Área do "+fg.getInformacao()+" : "+fg.calcularArea());
        }
    }
}
```

Testando a classe UsaFigurageometrica:



```
<terminated> UsaFiguraGeometrica [Java Application] C:\Pr
Entre com o raio do Círculo:
3
Entre com o lado do Quadrado:
4
Entre com o altura do Triângulo:
5
Entre com o base do Triângulo:
5
Área do Círculo: 28.274333882308138
Área do Quadrado: 16.0
Área do Triangulo: 12.5
```

4. Realize a seguinte implementação:
- a) Implemente a classe `Funcionario` com os atributos `nome` e `metricula`. Esta classe deve conter os seguintes métodos abstratos `getProfissao()`, `calcularSalario()`;
 - b) Crie a classe `Professor` que estende `Funcionario` com os campos `qtdeHoras` e `vlHora` ambos do tipo `double`. Implemente todas os métodos da classe `Funcionario`, para calcular o salário do professor deve multiplicar a quantidade de horas pelo seu respectivo valor.
 - c) Crie a class `Vendedor` que estende `Funcioanrio` com os campos `vlVenda` e `comissao` ambos do tipo `double`. Implemente todos os métodos da classe `Funcionario`, para calcular o salario do vendedor deve ser calculado o percentual do valor total vendido pelo vendedor.
 - d) Crie a classe `Motorista` que estende de `Funcionario` com os campos `salarioFixo`, `qtdeHorasExtras` e `vlHora` todos são tipo `double`. Implemente todos os métodos da classe `Funcionario`, para calcular o salario do motorista some o valor do salario fixo com o valor da multiplicação entre quantidade de horas extras e o seu respectivo valor.
 - e) Implemente a classe `UsaFuncionario`, que deve conter um vetor do tipo `Funcionario` com 5 posições. O usuário deve escolher qual funcionário vai cadastrar (`Professor`, `Vendedor`, `Motorista`) em cada posição do vetor. Deve ser realizada a entrada de todos os valores e logo em seguida exibir os dados dos respectivos funcionários assim como o seu salario e profissão.

```
public abstract class Funcionario {  
  
    //para captura dos dados  
    protected Scanner sc = new Scanner(System.in);  
  
    //métodos que todo funcionario deve ter  
  
    //retorna a profissao  
    public abstract String getProfissao();  
  
    //calcula o salario com base nos atributos  
    public abstract Double calcularSalario();  
  
    //realiza o preenchimento dos dados  
    public abstract void preencher();  
  
}
```

```
public class Professor extends Funcionario {

    public double qtdeHoras;
    public double vlHoras;

    @Override
    public String getProfissao() {
        return "Professor";
    }

    @Override
    public Double calcularSalario() {
        return qtdeHoras* vlHoras;
    }

    @Override
    public void preencher() {
        qtdeHoras = Double.parseDouble(
            JOptionPane.showInputDialog(null,
                "Entre com a quantidade de horas.", "Cadastro Professor",
                JOptionPane.QUESTION_MESSAGE));
        vlHoras = Double.parseDouble(
            JOptionPane.showInputDialog(null,
                "Entre com o valor de horas.", "Cadastro Professor",
                JOptionPane.QUESTION_MESSAGE));
    }
}

public class Vendedor extends Funcionario {

    public double vl venda, comissao;

    @Override
    public String getProfissao() {
        return "Vendedor";
    }

    @Override
    public Double calcularSalario() {
        return (vl venda*comissao)/100;
    }

    @Override
    public void preencher() {
        vl venda = Double.parseDouble(
            JOptionPane.showInputDialog(null,
                "Entre com o valor da venda.", "Cadastro Vendedor",
                JOptionPane.QUESTION_MESSAGE));
        comissao = Double.parseDouble(
            JOptionPane.showInputDialog(null,
                "Entre com a comissão.", "Cadastro Vendedor",
                JOptionPane.QUESTION_MESSAGE));
    }
}
```



```

public class Motorista extends Funcionario {

    private double salarioFixo, qtdeHorasExtras, vlHora;

    @Override
    public String getProfissao() {
        return "Motorista";
    }

    @Override
    public Double calcularSalario() {
        return salarioFixo + (vlHora*qtdeHorasExtras);
    }

    @Override
    public void preencher() {
        salarioFixo = Double.parseDouble(
            JOptionPane.showInputDialog(null,
                "Entre com o salarioFixo.", "Cadastro Motorista",
                JOptionPane.QUESTION_MESSAGE));
        vlHora = Double.parseDouble(
            JOptionPane.showInputDialog(null,
                "Entre com a quantidade de horas trabalhadas.", "Cadastro Motorista",
                JOptionPane.QUESTION_MESSAGE));
        qtdeHorasExtras = Double.parseDouble(
            JOptionPane.showInputDialog(null,
                "Entre com a quantidade de horas extras.", "Cadastro Motorista",
                JOptionPane.QUESTION_MESSAGE));
    }
}

public class UsaFuncionario {

    public static void main(String[] args) {
        //criamos um funcionario
        Funcionario[] funcionarios = new Funcionario[4];
        //vetor das opções pra o nosso InputDialog
        String[] opcoes = {"Professor", "Motorista", "Vendedor"};

        String op;
        for(int i=0; i< funcionarios.length; i++){
            //recebendo a opção desejada
            op = (String) JOptionPane.showInputDialog(null,
                "Entre com uma opção.", "Cadastro Funcionario",
                JOptionPane.QUESTION_MESSAGE, null, opcoes, "Professor");

            //verificando a opção
            if(opcoes[0].equals(op)) funcionarios[i] = new Professor();
            else if(opcoes[1].equals(op)) funcionarios[i] = new Motorista();
            else if(opcoes[2].equals(op)) funcionarios[i] = new Vendedor();

            //preenchendo o funcionario.
            funcionarios[i].preencher();
        }

        //mostrando os funcionario
        op="";
        for(Funcionario f : funcionarios){
            op+= f.getProfissao()+" , ganhe R$"+f.calcularSalario()+"\n";
        }
    }
}

```

```
JOptionPane.showMessageDialog(null,op);  
}  
}
```

“A invenção mais importante do século 19 foi a invenção do método de invenção.” – Alfred North

Concorrência – Threads

Whitehead

Seria interessante se pudéssemos fazer uma coisa por vez, e fazê-la bem, mas em geral isso é difícil. O corpo humano realiza uma grande variedade de operações paralelamente - ou, como diremos por todo este capítulo, concorrentemente. A respiração, a circulação sanguínea, a digestão, o pensamento e a locomoção, por exemplo, podem ocorrer simultaneamente. Todos os sentidos - visão, tato, olfato, paladar e audição - podem ocorrer ao mesmo tempo. Os computadores também realizam operações concorrentemente. É comum aos computadores pessoais compilar um programa, enviar um arquivo para uma impressora e receber mensagens de correio eletrônico em uma rede concorrentemente. Apenas os computadores que têm múltiplos processadores podem, de fato, executar operações concorrentemente. Em computadores de um único processador, os sistemas operacionais utilizam várias técnicas para simular a concorrência, mas em computadores como esses uma única operação pode executar por vez.

A maioria das linguagens de programação não permite que os programadores especifiquem atividades concorrentes. Em geral, elas fornecem apenas instruções de controle que permitem que os programadores realizem uma ação por vez, avançando para a próxima ação depois de a anterior ser concluída. Historicamente, a concorrência foi implementada com as primitivas de sistemas operacionais disponíveis apenas para programadores de sistemas experientes.

A linguagem de programação Ada, desenvolvida pelo Departamento de Defesa dos Estados Unidos tornou primitivos de concorrência amplamente disponíveis para as empresas contratadas do Departamento de Defesa que estavam construindo sistemas de controle e comando militar. Entretanto, a tecnologia Ada não foi amplamente utilizada em universidades e indústria comercial.

O Java disponibiliza a concorrência para o programador de aplicativos por meio de suas APIs. O programador especifica os aplicativos que contêm threads de execução, em que cada thread designa uma parte de um programa que pode executar concorrentemente com outras threads.

Essa capacidade, chamada **multithreading**, fornece capacidades poderosas para o programador de Java não disponíveis no núcleo das linguagens C e C++ em que o Java é baseado.

A programação de aplicativos concorrentes é um empreendimento difícil e propenso a erro. **Até mesmo alguns aplicativos concorrentes mais simples estão além da capacidade de programadores iniciantes.** Se achar que deve utilizar a sincronização em um programa, você deve seguir algumas diretrizes simples:

Primeiro, utilize classes existentes da API do Java que gerencia a sincronização para você. As classes na API do Java foram completamente testadas e depuradas e ajudam a evitar interrupções e armadilhas comuns.

Segundo, se achar que precisa de funcionalidades mais personalizadas do que aquelas fornecidas nas APIs do Java, você deve utilizar a palavra-chave `synchronized` e os métodos `wait`, `notify` e `notifyAll` do `Object`.

Por fim, se precisar de capacidade ainda mais complexa, então você deve utilizar as interfaces `Lock` e `Condition`.

As interfaces `Lock` e `Condition` são ferramentas avançadas e só devem ser utilizadas por programadores experientes que estão familiarizados com interrupções e armadilhas comuns da programação concorrente com a sincronização.

Classe `Thread`

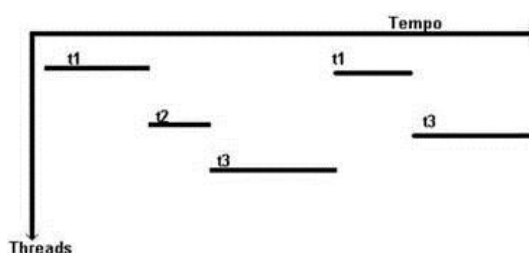
Para entender o funcionamento de uma thread é necessário analisar, inicialmente, um processo. A maioria dos sistemas de hoje são baseados em computadores com apenas um processador que executam várias tarefas simultâneas. Ou seja, vários processos que compartilham do uso da CPU tomando certas fatias de tempo para execução. A esta capacidade é denominado o termo multiprocessamento.

Teoricamente existe uma grande proteção para que um processo não afete a execução de outro, modificando, por exemplo, a área de dados do outro processo, a menos que haja um mecanismo de comunicação entre os processos

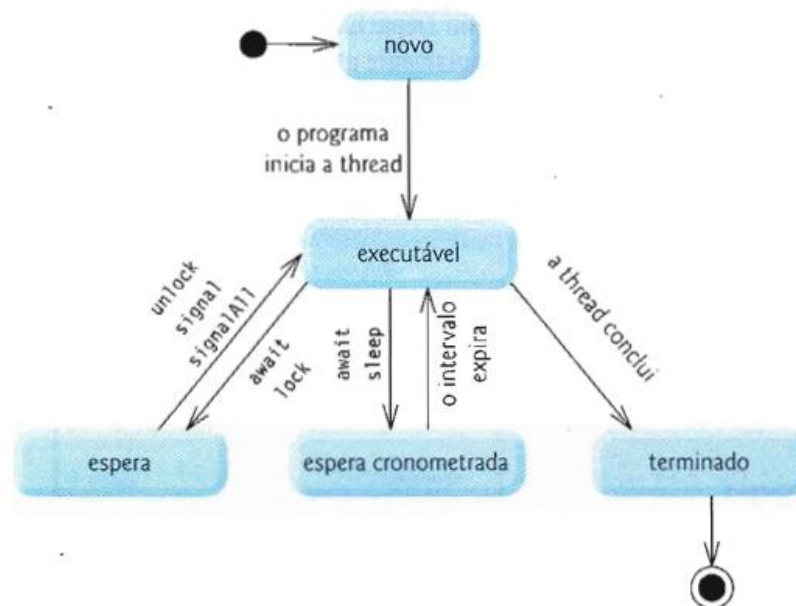
(IPC – Inter Process Communication). Este alto grau de isolamento reduz os desagradáveis GPFs (General Protection Fault), pois o sistema se torna mais robusto. Em contrapartida, o início de cada processo é bastante custoso, em termos de uso de memória e desempenho, e o mecanismo de troca de mensagens entre os processos é mais complexo e mais lento, se comparado a um único programa acessando a própria base de dados. Uma solução encontrada foi o uso de threads, (também conhecidas por linhas de execução).

A thread pode ser vista como um subprocesso de um processo, que permite compartilhar a sua área de dados com o programa ou outras threads. O início de execução de uma thread é muito mais rápido do que um processo, e o acesso a sua área de dados funciona como um único programa. Existem basicamente duas abordagens para a implementação das threads na JVM: utilização de mecanismos nativos de operação do S.O., e a implementação completa da operação thread na JVM.

A diferença básica é que as threads com mecanismos nativos do S.O. são mais rápidas. Em contrapartida a implementada pela JVM tem independência completa de plataforma. Basicamente, em ambos os casos, a operação das mesmas é obtida através de uma fatia de tempo fornecida pelo S.O. ou pela JVM. Isto cria um paralelismo virtual, como pode ser observado na figura abaixo, que representa a execução de três threads.



A execução de uma thread pode passar por quatro estados: novo, executável, bloqueado e encerrado.



Uma nova thread inicia seu ciclo de vida no estado **novo**. Ela permanece nesse estado até o programa iniciar a thread, o que a coloca no estado executável. Considera-se que uma thread nesse estado está executando sua tarefa.

Às vezes uma thread entra no estado de **espera** enquanto espera outra thread realizar uma tarefa. Uma vez nesse estado, a thread só volta ao estado executável quando outra thread sinalizar a thread de espera para retomar a execução.

Uma thread executável pode entrar no estado de **espera sincronizada** por um intervalo especificado de tempo. Uma thread nesse estado volta para o estado executável quando esse intervalo de tempo expira ou quando ocorre o evento que ele está esperando. As threads de espera sincronizada não podem utilizar um processador, mesmo que haja um disponível. Uma thread pode transitar para o estado de espera sincronizada se fornecer um intervalo de espera opcional quando ela estiver esperando outra thread realizar uma tarefa. Essa thread retornará ao estado executável quando ela for sinalizada por outra thread ou quando o intervalo sincronizado

expirar - o que ocorrer primeiro. Outra maneira de colocar uma thread no estado de espera sincronizada é colocá-la para dormir. Uma thread **adormecida** permanece no estado de espera sincronizada por um período designado de tempo (denominado **intervalo de adormecimento**) no ponto em que ele retoma para o estado executável. As threads dormem quando, por um breve período, não têm de realizar nenhuma tarefa. Por exemplo, um processador de texto pode conter uma thread que grave periodicamente uma cópia do documento atual no disco para fins de recuperação. Se a thread não dormisse entre os sucessivos backups, seria necessário um loop em que testaria continuamente se ela deve ou não gravar uma cópia do documento em disco. Esse loop consumiria tempo de processador sem realizar trabalho produtivo, reduzindo assim o desempenho de sistema. Nesse caso, é mais eficiente para a thread especificar um intervalo de adormecimento (igual ao período entre backups sucessivos) e entrar no estado de espera sincronizada. Essa thread retoma ao estado executável quando seu intervalo de adormecimento expira, ponto em que ela grava uma cópia do documento no disco e entra novamente no estado de espera sincronizada.

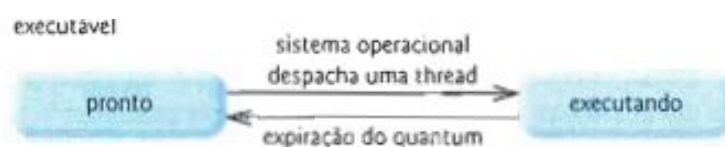
Uma thread executável entra no estado **terminado** quando completa sua tarefa ou, caso contrário, termina (talvez devido a uma condição de erro).

Quando uma thread entra pela primeira vez no estado executável a partir do estado novo, a thread está no estado **pronto**. Uma thread pronta entra no estado de execução (isto é, começa a executar) quando o sistema operacional atribui a thread a um processador - também conhecido como despachar a thread. Na maioria dos sistemas operacionais, cada thread recebe uma pequena quantidade de tempo de processador - denominada quantum ou fração de tempo - com o qual realiza sua tarefa. Quando o quantum da thread expirar, a thread retornará ao estado pronto e o sistema operacional atribuirá outra thread ao processador. As transições entre esses estados são tratadas unicamente pelo sistema operacional. A JVM não 'vê' esses dois estados - ela simplesmente visualiza uma thread quando ela está no estado executável e a deixa para o sistema operacional fazer a transição de threads entre os estados pronto e de execução. O processo que utiliza um sistema

operacional para decidir qual thread despachar é conhecido como agendamento de thread e depende das prioridades de thread.

Prioridades de thread e agendamento de thread

Cada thread Java tem uma prioridade que ajuda o sistema operacional a determinar a ordem em que as threads são agendadas. As prioridades do Java estão no intervalo entre MIN_PRIORITY (uma constante de 1) e MAX_PRIORITY (uma constante de 10). Informalmente, as threads com prioridade mais alta são mais importantes para um programa e devem ser alocadas em tempo de processador antes das threads de prioridade mais baixa. Entretanto, as prioridades de thread não podem garantir a ordem em que elas são executadas. Por padrão, cada thread recebe a prioridade NORM_PRIORITY (uma constante de 5). Cada thread nova herda a prioridade da thread que a criou.



Criando e executando threads

Em J2SE 5.0, o modo preferido de criar um aplicativo de múltiplas threads é implementar a interface Runnable (pacote java.lang) e utilizar classes e métodos predefinidos para criar Threads que executam os objetos Runnables. A interface Runnable declara um único método chamado run. Runnables são executados por um objeto de uma classe que implementa a interface Executor (pacote java.util.concurrent). Essa interface declara um único método chamado execute. Em geral, um objeto Executor cria e gerencia um grupo de threads denominado pool de threads. Essas threads executam os objetos Runnables passados para o método execute. O Executor atribui cada Runnable a uma das threads disponíveis no pool de threads. Se não houver nenhuma thread disponível no pool de threads) o Executor cria uma nova thread ou espera que uma se torne disponível para atribui a ela o Runnable que foi passado para método execute. Dependendo do tipo Executor, há um limite para o número de threads que podem ser criadas. A interface ExecutorService (pacote java.util.concurrent) é uma subinterface de Executor

que declara vários outros métodos para gerenciar o ciclo de vida do Executor. Um objeto que implementa a interface `ExecutorService` pode ser criado utilizando métodos static declarados na classe `Executors` (pacote `java.util.concurrent`).

Em nosso exemplo vamos criar um programa simples, uma contadora que implementa `Runnable`. Ela conta de 0 ate 50.

```
public class Contadora implements Runnable {  
    int num_thread; //representa o numero/nome da thread  
  
    public Contadora(int num_thread) {  
        super();  
        this.num_thread = num_thread;  
    }  
  
    @Override  
    public void run() {  
        for(int i=0; i<=50; i++)  
            System.out.println(num_thread+" - "+i);  
    }  
}
```

Colocamos um atributo inteiro para representar um “nome” para o objeto da classe `Contadora`. Com isso será possível visualizar melhor o que acontece no nosso teste.

```
public class TestaContadora {  
    public static void main(String args[]){  
        Contadora c = new Contadora(1); //nossa t  
        Thread t = new Thread(c);  
        t.start();  
  
        Contadora c2 = new Contadora(2); //nossa t  
        Thread t2 = new Thread(c2);  
        t2.start();  
    }  
}
```

Em nosso testador, vamos criar dois objetos da classe contadora e coloca-los em duas threads. Elas iram rodar quase que simultaneamente. Podemos perceber pelo resultado que não é possível saber quando cada uma será executada. Pode ocorrer diversos resultados diferentes.

Entra em cena o escalonador de threads.

O escalonador (scheduler), sabendo que apenas uma coisa pode ser executada de cada vez, pega todas as threads que precisam ser executadas e faz o processador ficar alternando a execução de cada uma delas. A ideia é executar um pouco de cada thread e fazer essa troca tão rapidamente que a impressão que fica é que as coisas estão sendo feitas ao mesmo tempo.

O escalonador é responsável por escolher qual a próxima thread a ser executada e fazer a troca de contexto (context switch). Ele primeiro salva o estado da execução da thread atual para depois poder retomar a execução da mesma. Aí ele restaura o estado da thread que vai ser executada e faz o processador continuar a execução desta. Depois de um certo tempo, esta thread é tirada do processador, seu estado (o contexto) é salvo e outra thread é colocada em execução. A troca de contexto é justamente as operações de salvar o contexto da thread atual e restaurar o da thread que vai ser executada em seguida.

Quando fazer a troca de contexto, por quanto tempo a thread vai rodar e qual vai ser a próxima thread a ser executada, são escolhas do escalonador. Nós não controlamos essas escolhas (embora possamos dar “dicas” ao escalonador). Por isso que nunca sabemos ao certo a ordem em que programas paralelos são executados.

Você pode pensar que é ruim não saber a ordem. Mas perceba que se a ordem importa para você, se é importante que determinada coisa seja feita antes de outra, então não estamos falando de execuções paralelas, mas sim de um programa sequencial normal (onde uma coisa é feita depois da outra, em uma sequência).

Todo esse processo é feito automaticamente pelo escalonador do Java (e, mais amplamente, pelo escalonador do sistema operacional). Para nós,

programadores das threads, é como se as coisas estivessem sendo executadas ao mesmo tempo.

Colocando threads para dormir

Em nosso exemplo, colocamos as nossas threads para dormir utilizando “Thread.sleep()” que recebe o tempo em milissegundos na qual a thread ficara dormindo.

```
public class Contadora implements Runnable {  
    int num_thread; //representa o numero/nome da thread  
  
    public Contadora(int num_thread) {  
        super();  
        this.num_thread = num_thread;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i <= 100; i++) {  
            try {  
                System.out.println(num_thread + " - " + i + ", now is sleeping..zz");  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

O método sleep deve estar dentro de um tratamento para a exception InterruptedException.

Existem outros métodos para controlar threads. Esse é um assunto bem avançado, não recomendado para iniciantes.

“Quantas maçãs não caíram na cabeça de Newton antes de ele ter percebido a dica!” – Robert Frost

Exercícios Complementares de Fixação

1) Escreva declarações, instruções ou comentários que realizam cada uma das tarefas a seguir:

- a. Declare as variáveis `c`, `thisIsAVariable`, `q76354` e `number` como sendo do tipo `int`;
- b. Solicite que o usuário insira um inteiro;
- c. Insira um inteiro e atribua o resultado á variável `int value`. Suponha que a variável `Scanner input` possa ser utilizada para ler um valor digitado pelo usuário.
- d. Se a variável `number` não for igual a 7, exiba “a variável não é igual a 7”.
- e. Imprima “Isso é um programa Java” em uma linha na janela de comando.
- f. Imprima “Isso é um programa Java” em duas linhas na janela de comando. A primeira linha deve terminar com “um”. Utilize o método `System.out.println`.
- g. Declare que um programa calculará o produto de três inteiros.
- h. Crie um `Scanner` que lê valores a partir da entrada-padrão.
- i. Declare as variáveis `x`, `y`, `z` e `result` como do tipo `int`.
- j. Solicite que o usuário insira o primeiro inteiro.
- k. Leia o primeiro inteiro digitado pelo usuário e armazene-o na variável `x`.
- l. Solicite que o usuário insira o segundo inteiro.
- m. Leia o segundo inteiro digitado pelo usuário e armazene-o na variável `y`.
- n. Solicite que o usuário insira o terceiro inteiro.
- o. Leia o terceiro inteiro digitado pelo usuário e armazene-o na variável `z`.
- p. Compute o produto dos três inteiros contidos nas variáveis `x`, `y`, `z` e atribua o resultado a variável `result`.
- q. Exiba a mensagem “o produto custa “ e exiba o valor da variável `result`.
- r. Some os inteiros ímpares entre 1 e 9 utilizando `for`. Assuma que as variáveis de inteiro `sum` e `count` foram declaradas.
- s. Calcule o valor 2.5 elevado á potência de 3, utilizando o método `Math.pow()`.
- t. Imprima os inteiros de 1 a 20, utilizando um loop `while` e a variável contadora `q`, assuma que a variável `i` foi declarada mas não inicializada. Imprima apenas 5 inteiros por linha.
- u. Repita a parte acima utilizando `for`, depois do.

- v. Atribua a soma de x e y e z e incremente x por 1 depois do calculo.
- w. Teste se a variável contador é maior que 10, se for imprima “contador é maior que 10”.
- x. Decrementa a variável x por 1, então subtraia o resultado da variável total.
- y. Calcule o resto após q ser dividido por divisor e atribua o resultado a q.
- z. Declare variáveis sum e x que serão do tipo int.
- aa. Atribua 1 a variável x.
- bb. Atribua 0 a variável sum.
- cc. Adicione a variável x a variável sum e atribua o resultado a variável sum.
- dd. Imprima “a soma é: “ seguido pelo valor da variável sum.

2) Escreva um aplicativo que solicita ao usuário inserir dois inteiros, obtém do usuário esses números e imprime sua soma, produto, diferença e quociente (divisão).

3) Escreva um aplicativo que insere três inteiros digitados pelo usuário e exibe a soma, a media, o produto e os números menores e maiores

4) Escreva um aplicativo que solicita ao usuário inserir dois inteiros, obtem do usuário esses números e exibe o numero maior seguido pelas palavras “é maior”. Se os números forem iguais, imprima “esses números são iguais”.

5) Escreva um aplicativo que le um inteiro e determina e imprime se ele é impar ou par.[dica: utilize o operador de modulo. Um número par é um múltiplo de 2. Qualquer múltiplo de 2 deixa um resto 0 quando dividido por 2]

6) Escreva um aplicativo que lê dois inteiros, determina se o primeiro é múltiplo do segundo e imprime o resultado.

7) Os motoristas se preocupam com a quilometragem dos seus automóveis. Um motorista monitorou vários tanques cheios de gasolina registrando a quilometragem dirigida e a quantidade de combustível em litros utilizados para cada tanque cheio. Desenvolva um aplicativo Java que receba como entrada os quilômetros dirigidos e os litros de gasolina consumidos para cada tanque cheio.

O programa deve calcular e exibir o consumo em quilometragem/litro para cada tanque cheio. Todos os cálculos de media devem ser produzir resultados de ponto flutuante. Utilize a classe Scanner e repetição

controlada por sentinela para obter os dados do usuário. Imprima todos os números de 150 a 300.

8) Imprima a soma de 1 até 1000.

9) Imprima todos os múltiplos de 3, entre 1 e 100.

10) Imprima os fatoriais de 1 a 10.

11) Escreva um programa que localize o menor de vários inteiros.

12) Escreva um aplicativo que calcule o produto dos inteiros ímpares de 1 a 15.

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

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

a. se x é par, $y = x / 2$

b. se x é ímpar, $y = 3 * x + 1$

c. imprime y

d. O programa deve então jogar o valor de y em x e continuar até que y tenha o valor final de 1. Por exemplo, para $x = 13$, a saída será: 40 -> 20 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1.

15) Imprima a seguinte tabela, usando fors encadeados:

a. 1

b. 4

c. 6 9

d. 8 12 16

e. $n \ n*2 \ n*3 \ \dots \ n*n$

16) Desenvolva um aplicativo Java que determinará se um cliente de uma loja de departamentos excedeu o limite de crédito em uma conta corrente. Para cada cliente, os seguintes fatos estão disponíveis:

Numero da conta

Saldo no início do mês

Total de todos os itens cobrados desse cliente no mês

Total de créditos aplicados ao cliente no mês

Limite de credito autorizado

O programa deve inserir todos esses fatos como inteiros, calcular o novo saldo (= saldo inicial + despesas – créditos), exibir o novo saldo e determinar se o novo saldo excede o limite de crédito do cliente. Para clientes cujo limite de credito for excedido, o programa devera exibir a mensagem “limite de credito excedido”.

17) Uma grande empresa paga seu pessoal de vendas com base em comissões. O pessoal recebe \$200 por semana mais 9% de suas vendas brutas durante essa semana. Por exemplo, um vendedor que realiza um total de vendas de mercadorias de R\$ 5.000 em uma semana recebe \$200 mais 9% de R\$5.000 ou um total de \$650. Foi-lhe fornecida uma lista dos itens vendidos por cada vendedor. Os valores desses itens são como segue:

Item	value
1	239.99
2	129.75
3	99,95
4	350,89

Desenvolva um aplicativo Java que receba a entrada do itens por um vendedor durante a ultima semana e calcule e exibe os rendimentos do vendedor. Não há limites quanto ao numero de itens que podem ser vendido por um mesmo vendedor.

18) Desenvolva um aplicativo Java que determine o salario bruto de cada um dos três empregados. A empresa paga “hora normal” pelas primeiras 40 horas trabalhadas por cada funcionário e 50% a mais para todas as horas trabalhadas além de 40 horas. Você recebe uma lista dos empregados da empresa, o numero de horas trabalhadas por empregado na ultima semana e o salário hora de cada empregado. Seu programa deve aceitar a entrada dessas informações para cada empregado e então determinar e exibir o salario bruto do empregado. Utilize a classes Scanner para inserir os dados.

19) O processo de localizar o maior valor (isto é, o valor máximo de um grupo de valores) é frequentemente utilizado em aplicativos de computador. Por exemplo, um programa de determina o vencedor de uma competição de vendas inseriria o numero de unidades vendidas por cada vendedor. O vendedor que vende mais unidades ganha a competição. Escreva um programa em Java

que aceite como entrada uma serie de 10 inteiros e determine e imprima o maior valor dos inteiros. Seu programa devera utilizar pelo menos três variáveis descritas a seguir:

a. Conter: um contador para contar até 10 (isto é, monitorar quantos números foram inseridos e determinar quando todos os 10 números foram processados).

b. Number: o inteiro mais recente inserido pelo usuário.

c. Largest: o maior numero encontrado até agora.

20) (desafio – palíndromos) um palíndromo é uma sequencia de caracteres que é lida da esquerda para direita ou da direita para esquerda. Por exemplo, cada um dos seguintes inteiros de cinco dígitos é um palíndromo: 12321, 55555, 45554 e 11611. Escreva um aplicativo que leia em um inteiro de cinco dígitos e determine se ele é um palíndromo. Se o numero não for de cinco dígitos, exiba uma mensagem de erro e permita que o usuário insira um novo valor.

21) Escreva um aplicativo que aceite como entrada um inteiro contendo somente 0s e 1s(isto é, um numero binário) e imprime seu equivalente decimal.[dica: utilize os operadores de resto e divisão para pegar os dígitos do número binário um de cada vez, da direita para esquerda. No sistema de números decimais, o dígito mais a direita tem o valor posicional de 1 e o próximo dígito a esquerda tem um valor posicional de 10, depois 100, depois 1000 e assim por diante. O numero decimal 234 pode ser interpretado como $4*1 + 3*10 + 2*200$. No sistema de números binários, o dígito mais a direita tem o valor posicional de 1, o próximo dígito a esquerda tem um valor posicional de 2, depois 4, depois 8 e assim por diante. O equivalente decimal do binário 1101 é $1*1 + 0*2 + 1*4 + 1*8$, ou $1+0+4+8$ ou 13.]

22) Escreva um programa que leia três valores diferentes de zero inseridos pelo usuário e determine e imprima se eles poderiam representar os lados de um triangulo.

23) O fatorial de um inteiro não negativo n é escrito como $n!$ (pronuncia-se n fatorial) e é definido como segue:

$N! = n (n-1) (n-2) \dots 1$ (para valores de n maiores ou igual a 1)

E $n!=1$ (para $n=0$)

Por exemplo: $5!=5.4.3.2.1$, o que dá 120.

a. Escreva um aplicativo que leia inteiro não negativo, calcule e imprima seu fatorial.

b. Escreva um aplicativo que estime o valor da constante matemática e utilizando a formula: $E = 1 + 1/1! + 1/2! + 1/3! + \dots$

c. Escreva um aplicativo que computa o valor de e^x utilizando a formula: $e^x = 1 + x/1! + x^2/2! + x^3/3! + \dots$

-----Exercícios a partir daqui utilizam classes-----

24) Crie uma classe chamada Date que inclui três informações com variáveis de instancia - um mês (tipo int), um dia (tipo int) e um ano (tipo int). Sua classe deve ter um construtor que inicializa as três variáveis de instância e assumir que os valores fornecidos são corretos. Forneça um método set e um get para cada variável de instância. Forneça um método displayDate que exibe o mês, o dia e o ano separados por barras normais.

Escreva um aplicativo de teste chamado DateTest que demonstre as capacidades da classe Date.

25) Um estacionamento cobra uma taxa mínima de R\$2 para estacionar por até três horas. Um adicional de R\$ 0,50 por hora não necessariamente inteira é cobrado após as três primeiras horas. A carga máxima para qualquer dado período de 24 horas é R\$ 10. Suponha que nenhum carro fica estacionado por mais de 24 horas por vez. Escreva um aplicativo que calcule e exiba as taxas de estacionamento para cada cliente que estacionou nessa garagem ontem. Você deve inserir as horas de estacionamento para cada cliente. O programa deve exibir a cobrança para o cliente atual e calcular e exibir o total dos recibos de ontem. O programa deve utilizar o método calculateCharges para determinar a cobrança para cada cliente.

26) Os computadores estão desempenhando um crescente papel na educação. Escreva um programa que ajudará um estudante do ensino fundamental a aprender multiplicação. Utilize um objeto Random para produzir dois inteiros positivos de um algarismo. O programa então deve fazer ao usuário uma pergunta, como **Quanto é 6 vezes 7?** O aluno insere então a resposta. Em seguida, o programa verifica a resposta do aluno. Se estiver correta, exiba a mensagem "Muito Bom!" e faça outra pergunta de multiplicação. Se a resposta estiver errada, exiba a mensagem "Não, por favor

tente de novo. " e deixe que o estudante tente a mesma pergunta repetidamente até por fim responder corretamente. Um método separado deve ser utilizado para gerar cada nova pergunta. Esse método deve ser chamado uma vez quando a aplicação inicia a execução e toda vez que o usuário responde à pergunta corretamente

27) O uso de computadores na educação é referido como instrução auxiliada por computador (CAI computer-assisted instruction). Um problema que se desenvolve em ambientes C AI é a fadiga do estudante. Esse problema pode ser eliminado variando as respostas do computador para prender a atenção do estudante. Modifique o programa do Exercício 26 de modo que os vários comentários sejam exibidos para cada resposta correta e cada resposta incorreta, como segue:

Respostas para uma resposta correta:

Very good! [Muito bom!]

Excellent! [Excelente!]

Nice work! [Bom trabalho!]

Keep up the good work! [Continue o bom trabalho!]

Respostas para uma resposta incorreta:

No. Please try again. [Não. Tente de novo.]

Wrong. Try once more. [Errado. Tente mais uma vez.]

Don't give up! [Não desista!]

No. Keep trying. [Não. Continue tentando.]

Utilize geração de números aleatórios para escolher um número entre 1 e 4 que será utilizado para selecionar uma resposta apropriada para cada resposta. Utilize uma instrução switch para emitir as respostas.

28) Sistemas mais sofisticados de instruções auxiliadas por computador monitoram o desempenho do estudante durante um período de tempo. A decisão sobre um novo tópico frequentemente é baseada no sucesso do estudante com tópicos prévios. Modifique o programa de Exercício 6 e 7 para contar o número de respostas corretas e incorretas digitadas pelo estudante. Depois que o aluno digitar 10 respostas, seu programa deve calcular a porcentagem de respostas corretas. Se a porcentagem for inferior a 75%, exiba "Peça ajuda ao seu instrutor" e reinicialize o programa para que outro estudante possa experimentá-lo.

29) Escreva um aplicativo que joga 'adivinha o número' como a seguir: Seu programa escolhe o número a ser adivinhado selecionando um inteiro aleatório no intervalo de 1 a 1.000. O aplicativo exibe o prompt "Guess a number between 1 and 1000" [Adivinhe um número entre 1 e 1.000]. O Jogador insere uma primeira suposição. Se o palpite do jogador estiver incorreto, seu programa deve exibir Too high. Try again [Muito alto. Tente novamente] ou Too low. Try again [Muito baixo. Tente novamente]. Para ajudar o jogador a 'zerar' mediante uma resposta correta, o programa deve solicitar ao usuário o próximo palpite. Quando o usuário insere a resposta correta, exiba Congratulations. You guessed the number! [Parabéns, você adivinhou o número !] e permita que o usuário escolha se quer jogar novamente.

30) Modifique o programa de Exercício 29 para contar o número de suposições que o jogador faz. Se o número for 10 ou menos, exiba Either you know the secret or you got lucky! [Você sabe o segredo ou tem muita sorte!] se o jogador adivinhar o número em 10 tentativas, exiba Aha! You know the secret ! [Aha! Você sabe o segredo!]. Se o jogador fizer mais que 10 adivinhações, exiba You should be able to do better! [Você é capaz de fazer melhor]. Porque esse jogo não deve precisar de mais que 10 suposições? Bem, acaba 'boa suposição' o jogador deve ser capaz de eliminar metade dos números. Agora mostre por que qualquer número de 1 a 1.000 pode ser adivinhado em 10 ou menos tentativas.

31) (Classe Retângulo) Crie uma classe Retangulo. A classe tem atributos length e width, cada um dos quais é configurado com o padrão 11. A classe deve ter métodos que calculam o perímetro (perimeter) e a área (area) do retângulo. A classe tem métodos set e get para o comprimento (length) e a largura (width). Os métodos set devem verificar se length e width são, cada um, números de ponto flutuante maiores que 0,0 e menores que 20,0. Escreva um programa para testar a classe Rectangle.

32) (Classe Conta de poupança) Crie uma classe SavingsAccount. Utilize uma variável static annualInterestRate para armazenar a taxa de juros anual para todos os correntistas. Cada objeto da classe contém uma variável de instância private savingsBalance para indicar a quantidade que o poupador atualmente tem em depósito. Forneça um método calculateMonthlyInterest para calcular os juros mensais multiplicando savingsBalance por

`annualInterestRate` e dividindo por 12 - esses juros devem ser adicionados a `savingsBalance`. Forneça um método `static modifyInterestRate` que configure `annualInterestRate` com um novo valor. Escreva um programa para testar a classe `SavingsAccount`. Instancie dois objetos `savingsAccount`, `saver1` e `saver2`, com saldos de \$ 2.000 e \$ 3.000, respectivamente. Configure `annualInterestRate` como 4% e então calcule o juro mensal e imprima os novos saldos para os dois poupadores. Em seguida, configure `annualInterestRate` para 5%, calcule a taxa do próximo mês e imprima os novos saldos para os dois poupadores.

Exercícios a partir daqui serão bem úteis ao seu aprendizado. Eles irão criar alguns mini projetinhos. A cada exercício esses projetinhos irá aumentar e adquirir novas funções e melhorias

33) Modele um funcionario. Ele deve ter o nome do funcionário, o departamento onde trabalha, seu salário (`double`), a data de entrada no banco (`String`), seu RG (`String`) e um valor booleano que indique se o funcionário ainda está ativa na empresa no momento ou se já foi mandado embora. Você deve criar alguns métodos de acordo com sua necessidade. Além deles, crie um método `bonifica` que aumenta o salario do funcionário de acordo com o parâmetro passado como argumento. Crie, também, um método `demite`, que não recebe parâmetro algum, só modifica o valor booleano indicando que o funcionário não trabalha mais aqui.

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

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

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. `Funcionario.class` será gerado, não podemos “executá-la” pois não há um `main`, mas assim verificamos que nossa classe `Funcionario` já está tomando forma.

Crie um método `toString()`, que não recebe nem devolve parâmetro algum e simplesmente imprime todos os atributos do nosso funcionário.

34) 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.setNome("Fiodor");
f1.setSalario(100);

Funcionario f2 = new Funcionario();
f2.setNome("Fiodor");
f2.setSalario(100);

System.out.println( f1 == f2 ? "iguais" : "diferentes");
```

35) 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.setNome("Fiodor");
f1.setSalario(100);

Funcionario f2 = f1;

System.out.println( f1 == f2 ? "iguais" : "diferentes");
```

O que acontece com o condicional do exercício anterior?

36) 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.

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

Modifique seu método `toString()` para que ele imprima o valor da `dataDeEntrada` daquele funcionário.

37) Crie uma classe `Empresa`. A `Empresa` tem um nome, `cnpj` e uma referência a uma lista de funcionário, além de outros atributos que você julgar necessário.

A `Empresa` deve ter um método `adiciona`, que recebe uma referência a funcionário como argumento, e guarda esse funcionário.

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 Funcionário e já passa a referência dele, que dentro do objeto possui rg, salário, etc.

Crie uma classe TestaEmpresa que possuirá um método main. Dentro dele crie algumas instâncias de Funcionário e passe para a empresa pelo método addFuncionario.

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

Em vez de mostrar apenas o salário de cada funcionário, você pode chamar o método toString() de cada Funcionário da sua lista.

```
public void mostraEmpregados(){
    for(Funcionario atual : funcionarios){
        if(atual.isAtivo()) System.out.println(atual.toString());
    }
}
```

39) Crie um método para verificar se um determinado Funcionário se encontra ou não como funcionário desta empresa.

```
public boolean contain(Funcionario f){
    for(Funcionario atual : funcionarios){
        if(atual.getNome().equals(f.getNome())) return true;
    }
    return false;
}
```

40) Faça com que sua classe funcionário possa receber, opcionalmente, o nome do funcionário 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 funcionário. Seria algo como:

```
public Funcionario(){} //construtor padrão

//construtor com nome
public Funcionario(String string) {
    this.nome=nome;
}
```

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

41) Adicione um atributo na classe funcionário de tipo int que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo funcionário. O primeiro funcionário instanciado tem identificador 1, o segundo 2, e assim por diante.

```
public class Funcionario {  
  
    private int id;  
    private String nome;  
    private String depto;  
    private double salario;  
    private Date dataEntrada;  
    private String rg;  
    private boolean ativo;  
  
    public Funcionario(){  
        id = this.hashCode();  
    }  
  
    public Funcionario(String string) {  
        this.nome=nome;  
        id = this.hashCode();  
    }  
  
    public int getId() {  
        return id;  
    }  
}
```

42) Como garantir que datas como 31/2/2005 não sejam aceitas pela sua classe Data? (não há necessidade de tratar anos bissextos)

43) 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)....)

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

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

45) 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 uma classe com método main e instancie essas classes, atualize-as e veja o resultado.

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

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

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

System.out.println(c.getSaldo());
System.out.println(cc.getSaldo());
System.out.println(cp.getSaldo());
```

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 e não de acordo com como nos referimos a ele.

46) 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.


```
public class AtualizadorDeContas {  
  
    private float saldoTotal = 0;  
    private float taxa;  
  
    public AtualizadorDeContas(float taxa) {  
        this.taxa = taxa;  
    }  
  
    void roda(Conta c) {  
        System.out.println("Saldo anterior: R$ "+c.saldo);  
        c.atualiza(taxa);  
        System.out.println("Saldo atual: R$ "+c.saldo);  
        saldoTotal+=c.saldo;  
    }  
  
    public float getSaldoTotal() {  
        return saldoTotal;  
    }  
  
    public void setSaldoTotal(float saldoTotal) {  
        this.saldoTotal = saldoTotal;  
    }  
}
```

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.depositar(1000);  
        cc.depositar(1000);  
        cp.depositar(1000);  
  
        AtualizadorDeContas adc = new AtualizadorDeContas(0.01f);  
        adc.roda(c);  
        adc.roda(cc);  
        adc.roda(cp);  
  
        System.out.println("Saldo Total: " + adc.getSaldoTotal());  
  
    }  
}
```

47) Crie um projeto interfaces e crie a interface AreaCalculavel:

```
public interface AreaCalculavel {  
    public float calculaArea();  
}
```

Queremos, agora, criar algumas classes que são AreaCalculavel:

```
public class Quadrado implements AreaCalculavel {  
  
    private int lado;  
  
    public Quadrado(int lado) {  
        this.lado = lado;  
    }  
  
    @Override  
    public float calculaArea() {  
        return this.lado * this.lado;  
    }  
}  
  
public class Retangulo implements AreaCalculavel {  
  
    private int largura;  
    private int altura;  
  
    public Retangulo(int largura, int altura) {  
        this.largura = largura;  
        this.altura = altura;  
    }  
  
    @Override  
    public float 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 e um Retângulo 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:

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

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

```
public interface Tributavel {  
  
    public float calculaTributos();  
  
}
```

Lemos essa interface da seguinte maneira: “todos que quiserem ser tributável precisam saber calcular tributos, devolvendo um float”.

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 quickfix do Eclipse vai sugerir que você reescreva o método; escolha essa opção e, depois, preencha o corpo do método adequadamente:

```
public class ContaCorrente extends Conta implements Tributavel {  
  
    public void depositar(float quantia){  
        if(quantia>0)  
            saldo+=quantia-0.10f;  
    }  
}
```

```
public void atualiza(float taxa){
    saldo+=saldo*taxa*2;
}

@Override
public float calculaTributos() {
    return this.getSaldo() * 0.01f;
}
}
```

Agora crie a classe SeguroDeVida, aproveitando novamente do Eclipse, para obter:

```
public class SeguroDeVida implements Tributavel {

    @Override
    public float calculaTributos() {
        return 42;
    }

}
```

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

```
public class TestaTributavel {

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

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

}
```

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.

49) Crie um GerenciadorDeImpostoDeRenda, que recebe todos os tributáveis de uma pessoa e soma seus valores, e inclua nele um método para devolver seu total:

```
public class GerenciadorDeImpostoDeRenda {  
    private float total;  
  
    public void adiciona(Tributavel t){  
        System.out.println("Adicionando tributável: " + t);  
        this.total += t.calculaTributos();  
    }  
  
    public float getTotal(){  
        return this.total;  
    }  
}
```

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

```
public class TestaGerenciadorDeImpostoDeRenda {  
    public static void main(String[] args) {  
        GerenciadorDeImpostoDeRenda gerenciador = new GerenciadorDeImpostoDeRenda();  
        SeguroDeVida sv = new SeguroDeVida();  
        gerenciador.adiciona(sv);  
        ContaCorrente cc = new ContaCorrente();  
        cc.depositar(1000);  
        gerenciador.adiciona(cc);  
        System.out.println(gerenciador.getTotal());  
    }  
}
```

Repare que, de dentro do GerenciadorDeImpostoDeRenda, você não pode acessar o método getSaldo, por exemplo, pois você não tem a garantia de que o Tributável que vai ser passado como argumento tem esse método. A única certeza que você tem é de que esse objeto tem os métodos declarados na interface Tributável.

É interessante enxergar que as interfaces (como aqui, no caso, Tributavel) costumam ligar classes muito distintas, unindo-as por uma característica que elas tem em comum. No nosso exemplo, SeguroDeVida e ContaCorrente são entidades completamente distintas, porém ambas possuem a característica de serem tributáveis.

Se amanhã o governo começar a tributar até mesmo PlanoDeCapitalizacao, basta que essa classe implemente a interface Tributavel! Repare no grau de desacoplamento que temos: a classe

GerenciadorDeImpostoDeRenda nem imagina que vai trabalhar como PlanoDeCapitalizacao. Para ela, o único fato que importa é que o objeto respeite o contrato de um tributável, isso é, a interface Tributavel. Novamente: programe voltado a interface, não a implementação.

50) (Adicionando Tratamento de Exceções) Na classe Conta, Modifique o método deposita(float x) e sacar(float quantia): Eles devem lançar uma exception chamada IllegalArgumentException, sempre que o valor passado como argumento for inválido (por exemplo, quando for negativo).

```
public void depositar(float quantia){
    if(quantia<0)
        throw new IllegalArgumentException
            ("Você tentou depositar um valor negativo");
    saldo +=quantia;
}

public void sacar(float quantia){
    if(quantia<0)
        throw new IllegalArgumentException
            ("Você tentou sacar um valor negativo");
    if(quantia>saldo)
        throw new IllegalArgumentException
            ("Você tentou sacar um valor acima do saldo");

    saldo-=quantia;
}
```

Crie uma classe TestaDeposita com o método main. Crie uma ContaPoupanca e tente depositar valores inválidos:

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

    Conta cp = new ContaPoupanca();
    try {
        cp.depositar(-100);
    } catch (IllegalArgumentException e) {
        System.out.println(e.getMessage());
    }

}
```

Apêndice 01 - Garbage collector

Coletor de lixo e o método finalize

Toda classe no Java contém os métodos da classe `Object` (pacote `java.lang`), um dos quais é o método **finalize**. Esse método é raramente utilizado. De fato, pesquisamos em mais de 6500 arquivos de código-fonte classes da API do Java e encontramos menos de 50 declarações do método `finalize`. Contudo, visto que esse método faz parte de cada classe, vamos discuti-lo aqui para ajudá-lo a entender seu propósito, caso você o encontre nos seus estudos ou na indústria. Os detalhes completos sobre o método `finalize` estão além do escopo deste livro e a maioria dos programadores não deve utilizá-lo - logo você verá por quê.

Todo objeto que você cria utiliza vários recursos do sistema, como a memória. Precisamos de uma maneira disciplinada de devolver recursos para o sistema quando eles são não mais necessários para evitar 'vazamentos de recurso'.

O Java Virtual Machine (JVM) realiza coleta de lixo automática para reivindicar a memória ocupada por objetos que não estão mais em uso. Quando não houver mais referências a um objeto, o objeto é marcado para coleta de lixo pela JVM. A memória desse objeto pode ser reivindicada quando a JVM executa seu coletor de lixo, responsável por recuperar a memória dos objetos que não são mais utilizados de modo que a memória possa ser utilizada para outros objetos. Portanto, vazamentos de memória que são comuns em outras linguagens como C e C++ (porque a memória não é automaticamente reivindicada nessas linguagens) são menos prováveis em Java (mas alguns ainda podem acontecer de maneiras sutis). Outros tipos de vazamentos de recursos podem ocorrer. Por exemplo, um aplicativo poderia abrir um arquivo no disco para modificar o conteúdo do arquivo. Se o aplicativo não fechar o arquivo, nenhum outro aplicativo poderá utilizar o arquivo até que o aplicativo que abriu o arquivo conclua.

O método `finalize` é chamado pelo coletor de lixo para realizar limpeza de terminação sobre um objeto um pouco antes de o coletor de lixo reivindicar a memória do objeto. O método `finalize` não recebe parâmetros e tem o tipo de retorno `void`. Um problema com relação ao método `finalize` é que não há garantias de o coletor de lixo executar em uma data/hora especificada. De fato, o coletor de lixo nunca pode executar antes de um programa terminar. Portanto, não fica claro se, ou quando, o método `finalize` será chamado. Por essa razão, a maioria dos programadores deve evitar o método `finalize`.

Apêndice 02 – Saída Formatada

Uma parte importante da solução de qualquer problema é a apresentação dos resultados.

Formatando a saída com printf

Formatação de saída precisa é alcançada com printf. O método printf pode realizar as capacidades de formatação como:

- Arredondar valores de ponto flutuante para um número indicado de casas decimais;
- Alinhar uma coluna de números com pontos de fração decimal aparecendo um em cima do outro;
- Alinhamento a direita e alinhamento a esquerda das saídas;
- Inserir caracteres literais em locais precisos em uma linha de saída;
- Representar números de ponto flutuante em formato exponencial;
- Representar inteiros em formato octal e hexadecimal;
- Exibir todos os tipo de dados com campos de largura de tamanho fixo e precisão;
- Exibir datas e horas em vários formatos.

Sintaxe:

System.out.printf(<string formatada>, <lista de argumentos>);

Ex: System.out.printf("x = %8.2f", x);

Parâmetros de formato

Sintaxe do especificador de formato:

% [índice][flags][dimensão][decimal] conversão

Índice: índice do argumento de substituição (opcional).

Flags: -,+,0. Alinhamento a esquerda, direita ou zeros a frente.

Dimensão: tamanho total: tamanho total reservado ao campo.

Decimal: número de casas decimais

Convenção: inteiro (d, para decimal; o, para octal; x, para hexadecimal);

– ponto flutuante (e , para PF em notação exponencial, f, para PF no formato decimal; g, para PF em decimal ou exponencial com base na magnitude do valor; a, para PF no formato hexadecimal).

- Strings (s) e char (c).

Imprimindo datas e horas

Com o caractere de conversão t ou T, podemos imprimir datas e horas em vários formatos. O caractere de conversão t ou T sempre é seguido por um caractere de sufixo de conversão que especifica o formato de datas e/ou horas.

- C: exibe a data e hora como: <dia mês data hora:minutos:segundos fusohorario ano>
- F: data : ano-mês-dia
- D: data: mês/dia/ano
- r : hora:minuto:segundo am/pm
- R: hora:minuto
- A: nome do dia da semana
- B: nome do mês
- d: dia com dois dígitos
- m: mês com dois dígitos
- y: ano com quatro dígitos
- H e i: horas no relógio de 24 e 12 horas respectivamente
- M: minutos
- S: segundos
- Z: fuso horário
- P: marcador de manhã ou tarde

Largura e precisão de campos

O tamanho exato de um campo em que são impressos é especificado por uma largura de campo. O programador insere um inteiro que representa a largura de campo entre o sinal de porcentagem e o caractere de conversão.

Apêndice 03 - Os 10 mandamentos do bom programador Java

Escrever bom código é o primeiro passo para ser bom programador. Existem algumas boas práticas relativas a como o código deve ser organizado, mas existem alguns princípios básicos que qualquer programador – por mais inexperiente que seja – deve saber e seguir.

1. Conheça e use as bibliotecas padrão

Existe um imenso ganho em conhecer a API padrão. Sobre tudo os pacotes `java.util`, `java.util.concurrent` e `java.io`. A API de coleções é extremamente importante.

Conhecer a API padrão não só lhe dará um conhecimento profundo da linguagem e da plataforma mas também o poupará de implementar algoritmos complexos. Use os que a API oferece porque eles foram criados por engenheiros experientes e testados por milhares de pessoas em todo o mundo.

2. Minimize a acessibilidade dos membros das suas classes

Isto é simples. Declare todos os atributos de estado como `private`, mesmo quando você decidir que subclasses devem acesso a eles, nunca os declare `protected`. Declare métodos `protected` em vez. Declare os construtores privados e forneça métodos estáticos para construir o objeto (métodos-fábrica). Não coloque métodos modificadores (`set`) a não ser que precise deles. Coloque apenas os acessores (`get`). Para inicializar o objeto passe os atributos no construtor ou no método estático de fábrica.

As vantagens são inúmeras porque você estará aumentando a possibilidade de encapsular comportamento ao mesmo tempo que diminui a superfície exposta da sua classe.

3. Prefira composição em vez de herança

Se o seu objeto parece ser uma lista, não o faça estender `List`, em vez faça-o ter um `List` como atributo interno. Se o seu objeto parece ser um usuário, talvez seja melhor que ele tenha um atributo usuário internamente.

Herança é um recurso poderoso, mas escasso. Uma vez que tenha usado o seu recurso de herança não mais o poderá alterar. Nunca. Portanto, adie o uso de herança o máximo possível.

4. Implemente criteriosamente o método `equals` para os seus objetos. Implemente também `hashCode` e `toString`

Implementar equals é um pouco mais complexo do que parece. Sempre que equals for sobrescrito você precisa também sobrescrever hashCode já que dois estão relacionados.toString é bom implementar porque é muito útil para debug e conversões do objeto paraString.

5. Use Enum em vez de int

Utilize Enum em vez de constantes definidas com int. O tipo Enum foi criado exatamente para isso, então use-o em seu benefício.

O Enum tem ainda a curiosa propriedade de poder servir para criar um verdadeiro Singleton em Java, contudo, evite singletons e prefira o uso do padrão Registry.

6. Prefira Interfaces para definir tipos, e apenas para isso.

Ao programar sempre defina as suas variáveis, retorno de métodos e parâmetros de métodos com interfaces. Ao definir hierarquias sempre comece por definir uma interface. Prefira a interface à classe abstrata. Utilize a classe abstrata como uma implementação padrão/ utilitária da interface, mas não para definir o tipo do objeto.

Nunca utilize interfaces para definir constantes. Isso é uma prática obsoleta.

Enum para esse tipo de funcionalidade.

7. Lide com exceções da forma certa

Não semeie código try-catch pelo seu sistema à toa. Faça-o apenas na fronteira da camada e use-o da forma certa. Não logue todas as exceções que se lembrar em todos os cantos do código. Deixe o try-catch na fronteira da camada tratar disso.

Use exceções não-verificadas a menos que esteja desenhando uma API ou camada que será usada por outros em muitos projetos. Use exceções verificadas sempre que a API estiver utilizando recursos de hardware, do sistema operacional ou outros que consumam recursos e/ou sejam lentos de inicializar (por exemplo, threads e conexões de rede)

Nunca, nunca, ignore exceções. Nunca faça isso. Nunca.

Sempre documente as exceções que os seus métodos lançam. Que sejam de validação de parâmetros, quer sejam relativas à lógica interna do método.

8. Sempre verifique a validade dos parâmetros que recebe

Ao definir um método, defina que valores são válidos e quais não são válidos para cada parâmetro. Sempre que encontrar algum parâmetro inválido interrompa o funcionamento do método lançando uma exceção.

Faça a verificação no início do método para evitar alocar recursos que não serão usados e para evitar verificações repetidas no meio do código.

Mesmo quando os parâmetros veem de outros pontos do seu sistema, sempre faça a verificação. Encare isso como uma medida de segurança e não como um estorvo. Por exemplo, em um sistema web, mesmo que um código javascript tenha validado os campos do cadastro, faça uma nova verificação do lado do servidor.

9. Use o tipo certo para o trabalho

Não use `double` ou `float` a menos que saiba o que está fazendo. Sobretudo nunca use estes tipos para conter dados que representem valores monetários. Como 0.1 e 0.01 não são representáveis desta forma, mas são muito comuns em quantidades financeiras o seu sistema está votado a criar erros de arredondamento, que podem significar perder dinheiro real. Não faça isso.

Se quer fazer contas com valores exatos utilize `BigDecimal`. Para trabalhar com dinheiro use o padrão `Money`.

Não utilize `Calendar` para representar ponto no tempo, use `Date` ou `long`. Use `Calendar` apenas para fazer cálculos ou conhecer informações específicas da data, como por exemplo qual foi o dia da semana.

Principalmente nunca use `String` quando outro tipo é mais apropriado. Se esse tipo não existe, crie-o. Dê uma olhada no padrão `Tiny Type` para o ajudar.

10. Saiba implementar `Comparable` e `Cloneable`

`Comparable` indica que o objeto representa algo que pode ser ordenável, tal como datas, textos e números podem. É útil sobretudo quando você sente a necessidade de ter ou apresentar os objetos dessa classe em ordem. Contudo só pode ser usado se o objeto apenas suporta uma única forma de ordenação. Por exemplo, datas só podem ser ordenados por ordem cronológica, mas clientes podem ser ordenados por nome, data de nascimento, número de comprar, valor das compras, etc.. Implemente `Comparable` apenas nas classes que só suportam um tipo de ordenação. Para os outros objetos (como cliente) pode criar classes à parte para cada tipo de comparação, que implementem `Comparator`.

`Cloneable` indica que o objeto pode ser copiado. É preciso ter cuidado ao implementar `Cloneable`. Você só deve implementar `Cloneable` quando tiver a certeza que isso não terá efeitos secundários. Por exemplo, se a sua classe não é final e implementar `Cloneable` as classes filhas terão que sobrescrever esse método. O problema é que elas podem se esquecer de fazer isso, e você terá o problema de ao clonar um objecto da classe filha obter um objeto da classe pai.

Em vez de Cloneable considere implementar um construtor de cópia. Um construtor de cópia é um construtor que recebe um objeto da mesma classe e copia o estado desse objeto. Aqui também é preciso ter muito cuidado para não fazer os dois objetos partilharem o mesmo estado. Em caso de dúvida não implemente Cloneable.

Apêndice 04 - Separação de Responsabilidades e Encapsulamento

Descrever os axiomas fundamentais da orientação a objetos em poucas palavras é uma tarefa difícil. Mas, ao mesmo tempo, vital. Não é de imaginar que alguém possa usar uma linguagem orientada a objetos (OO) sem os conhecer. Não menos importante que os conhecer é saber identificá-los e usá-los.

O primeiro, e talvez o único, princípio de OO é o da Separação de Responsabilidades. O termo em inglês *Separation of Concerns* (SoC) define melhor o significado. *Concerns* não é só a responsabilidade, mas também a preocupação: “Princípio de Separação de Preocupações” dá uma idéia melhor do que se quer dizer. Em resumo, um objeto deve fazer apenas uma tarefa e fazê-la bem. Não se deve *preocupar* com o que os outros objetos fazem. Num sistema macroscópico em que um objeto coordena objetos menores, ele confia que esses objetos executam bem a sua tarefa e não se *preocupa* com o como eles a executam. A única coisa que tem que existir entre todos os objetos é o consenso de com o quê, cada um, se deve *preocupar*.

A preocupação de cada objeto é apenas zelar pela melhor concretização da sua responsabilidade no sistema e zelar para que nada corrompa o controle que ele tem sobre como executar sua tarefa. Mas como conseguir a separação de responsabilidades? A única resposta a esta pergunta é: Encapsulamento. É através das diferentes formas de encapsulamento que o objeto concretiza seu objetivo.

1. Talvez a menos óbvia forma de encapsulamento é o próprio objeto em si. O fato de podermos juntar numa entidade só estado e comportamento é uma forma de encapsulamento.

2. Uma forma mais conhecida de encapsulamento é o nível de ocultação. Ou seja, o nível em que o objeto deixa visível, ou não, aos outros objetos aquilo que ele é, ou faz.

3. A herança é também uma forma de encapsulamento. O fato de um objeto se esconder sobre uma ou mais capas de identidade esconde dos outros objetos seu verdadeiro funcionamento e intenções.

Em Java estas três formas de encapsulamento e algumas regras simples permitem que o Princípio da Separação de Responsabilidades seja sempre possível de satisfazer. A própria definição da linguagem define os três artefatos necessários.

1. A **Classe** detém todos os conhecimentos sobre o objeto: como o criar e destruir, o que ele faz e o que é. A Classe é a tradução da primeira forma de encapsulamento.

2. Os **modificadores de acesso** (default, protected , private e public) instituem diferentes níveis de oclusão implementando assim, a segunda forma de encasupalmento.

3. A **Interface** dá corpo à terceira forma de encapsulamento, permitindo que um só objeto tenha várias caras e execute o mesmo trabalho sob condições diferentes.

Embora os artefatos embutidos na linguagem sejam os blocos fundamentais da construção de objetos e implantação do Principio de Separação de Responsabilidades existem diferentes combinações possíveis. O uso exaustivo dessas combinações levar-nos-á a encontrar um conjunto de padrões recorrentes que poderemos então usar diretamente na construção de nossos objetos: os Padrões de Projeto.

Apêndice 05 – Declarações e controle de Acesso

Identificadores legais

Ao declarar uma variável seu nome deverá começar com letra, cifrão (\$) ou caractere de conexão (_). Não podendo começar com números. Depois do primeiro caractere, os identificadores podem conter qualquer combinação de letras, caracteres de moedas, caracteres de conexão ou números.

Ex.: **Legais:** int a,\$c, __2_w, _\$, this_is_a_very_detailed_name;

Illegais: int b, -d, e#, .f, 7g;

Convenção de código

Concordar com um conjunto de padrões e programa-los ajuda a diminuir o esforço envolvido em testar, fazer a manutenção e melhorar qualquer código.

Classes e interfaces

A primeira letra deve ser maiúscula e, se várias palavras forem escritas juntas para formar o nome, a primeira letra de cada palavra deve ser maiúscula ("camelCase"). Os nomes de classes normalmente devem ser substantivos:

Ex: Dog //cachorro

Account // Conta

PrintWriter //Impressora

Para interfaces, os nomes normalmente devem ser adjetivos:

Ex: Runnable // Executável

Serializable //Serializável

Métodos

A primeira letra deve ser minúscula, e depois as regras camelCase devem ser usadas. Os nomes devem ser pares de verbo-substantivo.

Ex: getBalance // obterBalanço

doCalculation //fazerCalculo

setCustomerName // defineNomeDoCliente

variáveis

Como nos métodos. Usar nomes curtos e significativos.

ex: `buttonWidth // larguraDoBotão`

`accountBalance // balançoDaConta`

constantes

São criadas marcando-se variáveis como `static` e `final`, nomeadas usando-se letras maiúsculas com caracteres `underscore` como separadores.

ex: `MIN_HEIGHT //ALTURA_MINIMA`

Apêndice 06 – palavras reservadas

Uma das coisas que mais gosto na linguagem java é o seu pequeno conjunto de palavras reservadas. Até à versão 6 existem 53 palavras-chaves, das quais apenas 51 são realmente utilizadas. Só como exemplo o C# tem 78 palavras reservadas.

Modificadores de acesso

private: acesso apenas dentro da classe

protected: acesso por classes no mesmo pacote e subclasses

public: acesso de qualquer classe

Modificadores de classes, variáveis ou métodos

abstract: classe que não pode ser instanciada ou método que precisa ser implementado por uma subclasse não abstrata

class: especifica uma classe

extends: indica a superclasse que a subclasse está estendendo

final: impossibilita que uma classe seja estendida, que um método seja sobrescrito ou que uma variável seja reiniciada

implements: indica as interfaces que uma classe irá implementar

interface: especifica uma interface

native: indica que um método está escrito em uma linguagem dependente de plataforma, como o C

new: instancia um novo objeto, chamando seu construtor

static: faz um método ou variável pertencer à classe ao invés de às instâncias

strictfp: usado em frente a um método ou classe para indicar que os números de ponto flutuante seguirão as regras de ponto flutuante em todas as expressões

synchronized: indica que um método só pode ser acessado por uma thread de cada vez

transient: impede a serialização de campos

volatile: indica que uma variável pode ser alterada durante o uso de threads

Controle de fluxo dentro de um bloco de código

break: sai do bloco de código em que ele está

case: executa um bloco de código dependendo do teste do *switch*

continue: pula a execução do código que viria após essa linha e vai para a próxima passagem do loop

default: executa esse bloco de código caso nenhum dos testes de switch-case seja verdadeiro

do: executa um bloco de código uma vez, e então realiza um teste em conjunto com o *while* para determinar se o bloco deverá ser executado novamente

else: executa um bloco de código alternativo caso o teste *if* seja falso

for: usado para realizar um loop condicional de um bloco de código

if: usado para realizar um teste lógico de verdadeiro ou falso

instanceof: determina se um objeto é uma instância de determinada classe, superclasse ou interface

return: retorna de um método sem executar qualquer código que venha depois desta linha (também pode retornar uma variável)

switch: indica a variável a ser comparada nas expressões *case*

while: executa um bloco de código repetidamente até que uma certa condição seja verdadeira

Tratamento de erros

assert: testa uma expressão condicional para verificar uma suposição do programador

catch: declara o bloco de código usado para tratar uma exceção

finally: bloco de código, após um *try-catch*, que é executado independentemente do fluxo de programa seguido ao lidar com uma exceção

throw: usado para passar uma exceção para o método que o chamou

throws: indica que um método pode passar uma exceção para o método que o chamou

try: bloco de código que tentará ser executado, mas que pode causar uma exceção

Controle de pacotes

import: importa pacotes ou classes para dentro do código

package: especifica a que pacote todas as classes de um arquivo pertencem

Primitivos

boolean: um valor indicando verdadeiro ou falso

byte: um inteiro de 8 bits (signed)

char: um caracter unicode (16-bit unsigned)

double: um número de ponto flutuante de 64 bits (signed)

float: um número de ponto flutuante de 32 bits (signed)

int: um inteiro de 32 bits (signed)

long: um inteiro de 64 bits (signed)

short: um inteiro de 16 bits (signed)

Variáveis de referência

super: refere-se a superclasse imediata

this: refere-se a instância atual do objeto

Retorno de um método

void: indica que o método não tem retorno

Palavras reservadas não utilizadas

const: Não utilize para declarar constantes; use `public static final`

goto: não implementada na linguagem Java por ser considerada prejudicial

Literais reservados

De acordo com a Java Language Specification, **null**, **true** e **false** são tecnicamente chamados de valores literais, e não *keywords*. Se você tentar criar algum identificador com estes valores, você também terá um erro de compilação.