Estratégia
CONCURSOS

Aul

Professor: Equipe Informática e TI, Judah Reis, Pedro Henrique Chagas Freitas

Sumário

1 JAVA: CONCEITOS BÁSICOS

Vamos falar agora sobre uma das linguagens mais famosas do mundo! *Professor, o que é Java?* É uma linguagem de programação orientada a objetos, multiplataforma, robusta, portável, segura, extensível, concorrente e distribuída. *E ela é totalmente orientada a objetos?* Não! *Por que não?* Porque nem todos os seus tipos de dados são objetos (possui alguns tipos primitivos: **int**, **float**, **long**, **double**, **char**, etc).

Só por causa disso, professor? Não, ela também não suporta Herança Múltipla! *Entendi, mas existe alguma linguagem totalmente orientada a objetos?* Sim, por exemplo: Smalltalk! *E ela é uma linguagem compilada ou interpretada?* Na verdade, ela é híbrida, i.e., ela é compilada e interpretada! *Professor, ouvi dizer que Java é lento! É verdade?* Atualmente, não!

De fato, era lenta no início! No entanto, na década passada houve diversas melhorias na Java Virtual Machine (JVM) e em seu Compilador JIT[1]. Hoje em dia, o desempenho geral do Java é absurdamente rápido de acordo com diversos benchmarks. A Máquina Virtual é capaz de realizar diversas otimizações por meio de algoritmos heurísticos e o Compilador é capaz de identificar hotspots.

Vocês sabiam que Java é uma Linguagem WORA? Pois é, esse acrônimo significa *Write Once, Run Anywhere* ou *Escreva uma vez, execute em qualquer lugar*. Trata-se de um slogan para exemplificar os benefícios multiplataforma da linguagem Java! Idealmente, isso significa que um programa em Java (uma vez compilado em um *bytecode*) pode rodar em qualquer equipamento que possua uma JVM!

A linguagem Java foi criada pela *Sun Microsystems*, que depois foi adquirida pela *Oracle*, por quem é mantida atualmente. *Por que ela é uma linguagem concisa e simples?* Porque não contém redundâncias e é fácil de entender, implementar e utilizar. Ela possui sintaxe bastante parecida com C++, facilitando o aprendizado e a migração por novos programadores.

Professor, Java é robusta? Sim! Além de ser fortemente tipada, foi desenvolvida para garantir a criação de programas altamente confiáveis. Ela não dispensa uma programação cuidadosa, porém elimina alguns tipos de erros de programação possíveis em outras linguagens. A ausência da aritmética de ponteiros também exclui toda uma classe de erros relacionados a esse tipo de estrutura.

[1] Compilador Just-In-Time (JIT) é o compilador que altera a maneira na qual programas em Java são executados, geralmente otimizando-os e tornando-os mais rápidos.

O acesso a *arrays* e *strings*, e a conversão de tipos são checados em tempo de execução para assegurar a sua validade. O *Garbage Collector*[2] faz a desalocação automática de memória evitando, erros de referência e desperdício de memória. Finalmente, o recurso de Exception Handling permite o tratamento de erros em tempo de execução, por um mecanismo robusto, análogo ao do C++.

Java é também uma linguagem portável e multiplataforma! O Compilador é capaz de gerar um código intermediário (bytecode), que permite que o mesmo programa possa ser executado em qualquer máquina ou sistema operacional que possua uma JVM. Ademais, busca que todos os aspectos da linguagem sejam independentes de plataforma (Ex: ela especifica o tamanho e comportamento de cada tipo de dado).

Dessa forma, aplicações funcionam da mesma maneira em qualquer ambiente. Podemos dizer que Java é uma linguagem concorrente ou *multithreaded*, i.e., pode realizar diversas tarefas assincronamente com o uso de threads, que são suportadas de modo nativo. Java torna a manipulação de threads tão simples quanto trabalhar com qualquer variável.

Java é uma linguagem distribuída, i.e., foi projetada para trabalhar em um ambiente de redes, oferecendo bibliotecas para facilitar a comunicação, manipulando objetos distribuídos e oferecendo suporte à conectividade (Ex: URL, Sockets, Protocolos, etc). Ela também é uma linguagem segura, implementando encapsulamento, restringindo o acesso e a execução de diversos programas, tratando exceções, etc.

Além disso, possui um verificador de bytecodes, que investiga e procura códigos maliciosos que eventualmente podem ter sido inseridos, rompendo com a integridade dos dados. Por fim, ele também possui o *Security Manager*, utilizado para impedir, por exemplo, que applets executem códigos arbitrariamente. Isso impede o acesso direto a informações pela memória ou inserir código estranho.

Para finalizar, cabe salientar que Java é uma linguagem absurdamente extensível. *Por que, professor?* Porque ela integra diversas bibliotecas com o código nativo, além de permitir o carregamento dinâmico de classes em tempo de execução. Em outras palavras, os programas são formados por uma coleção de classes armazenadas independentemente e que podem ser carregadas no momento de utilização.

1.1 JAVA: PLATAFORMA JAVA

[2] Garbage Collector (ou Coletor de Lixo) é o responsável pela automação do gerenciamento de memória. Ele é capaz recuperar uma área de memória inutilizada por um programa, evitando vazamento de memória.

Java é tanto uma plataforma quanto uma linguagem de programação orientada a objetos que permite o desenvolvimento de aplicações em diversas plataformas diferentes. Como já foi dito anteriormente, Java está presente desde dispositivos pequenos (Smartphone, Tablet, etc) a máquinas de grande porte (Servidores, Mainframes, etc). A linguagem Java possui cinco ambientes de desenvolvimento:

- Java Standard Edition (Java SE): trata-se de 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 (JVM), Compilador (Javac), Bibliotecas (APIs), entre outras ferramentas. Em geral, rodam em computadores pessoais, notebooks, etc.

- Java Enterprise Edition (Java EE): trata-se do padrão para desenvolvimento de sistemas corporativos, voltada para aplicações multicamadas, baseadas em componentes executados em servidores de aplicações – ele inclui o Java SE. Contém bibliotecas para acesso a base de dados, RPC, CORBA, entre outras. As aplicações podem ou não estar na internet.

- Java Micro Edition (Java ME): trata-se do padrão aplicado a dispositivos compactos ou móveis, como smartphones, tablets, controles remotos, etc. Permite o desenvolvimento de softwares embarcados, i.e., aplicações que rodam em um dispositivo de propósito específico, desempenhando alguma tarefa útil. Em geral, possuem limitações de recursos como memória ou processamento.

- Java Card: tecnologia que permite que pequenos aplicativos baseados em Java (conhecidos como Applets) sejam executados com segurança em smartcards e outros dispositivos similares com grandes limitações de processamento e armazenamento. *Sabe o chip do seu celular?* Java Card! *Sabe o chip do cartão de crédito?* Java Card!

- Java FX: trata-se de uma plataforma de software multimídia para a criação e disponibilização de Rich Internet Application (RIA) que pode ser executada em diversos dispositivos diferentes. Ele permite a criação de aplicações ricas para navegadores, smartphones, televisores, video-games, blu-rays, etc – são os menos conhecidos.

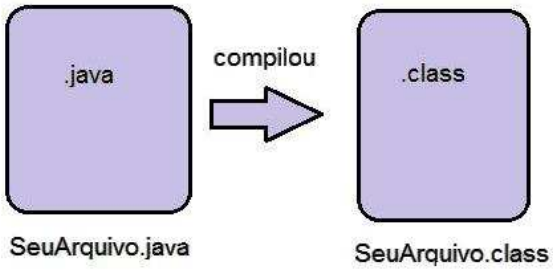
Um programa escrito para a plataforma Java necessita de um ambiente de execução chamado Java Runtime Environment (JRE)! *O que tem nesse negócio, professor?* Ele contém uma Máquina Virtual (JVM) e Bibiliotecas (APIs). *E o Java Development Kit*

(JDK)? Bem, eles contêm a JRE e outros componentes úteis para executar aplicações (Exemplo: Javac, Javadoc, Jar, Appletviewer, Jconsole, Jstack, Jhat, etc).

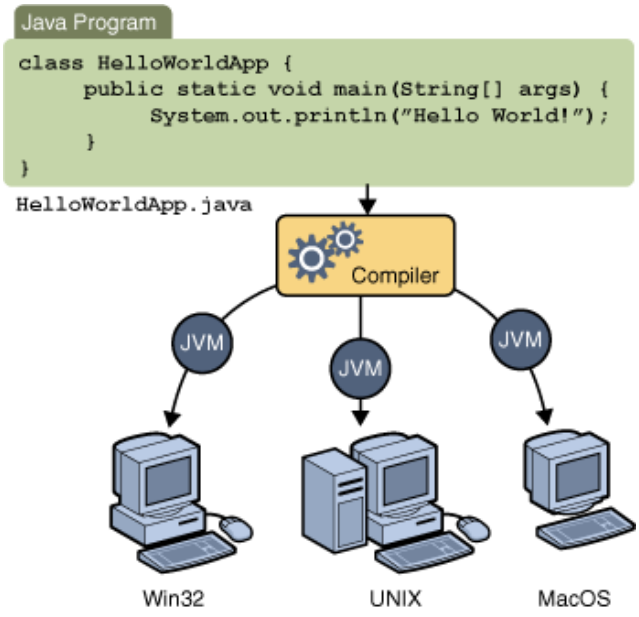
Portanto, é o seguinte: se você deseja somente executar alguma aplicação Java no seu computador ou navegador, basta instalar um JRE! No entanto, se você planeja programar em Java, você precisará de um JDK (que contém a JRE)! *Entenderam?* É bastante simples! JRE é o mínimo que você precisa para rodar uma aplicação e o JDK é o que você precisa para desenvolver uma aplicação!

1.2 JAVA: COMPILAÇÃO E INTERPRETAÇÃO

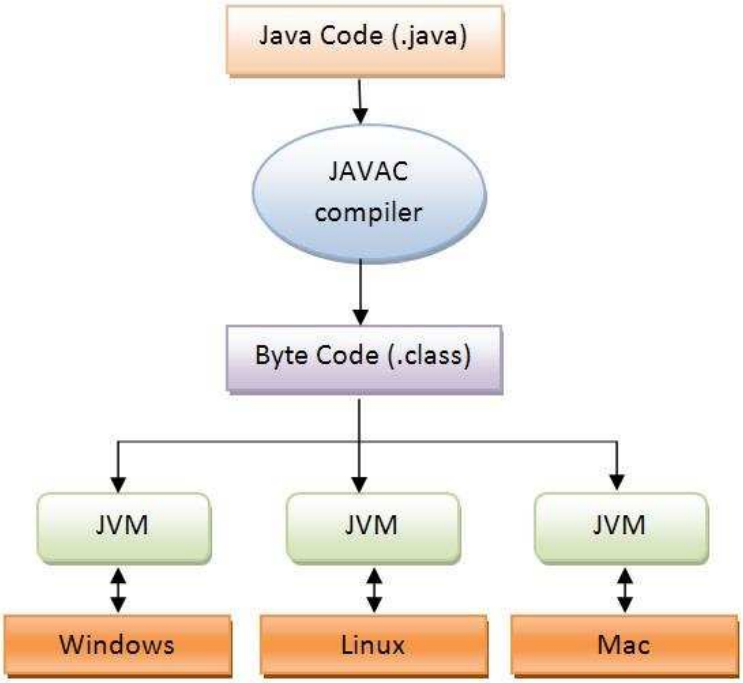
A Linguagem Java tem dois processos de execução de código-fonte: Compilação e Interpretação! Vamos lá... o programador escreve um código em Java em um editor de texto, por exemplo. Ele salva com a extensão **.java** e passa por um compilador (JavaC)! Esse compilador transforma o arquivo **.java** em código de máquina e em um arquivo **.class**, também chamado bytecode – como mostra a imagem abaixo.



O bytecode é um código intermediário, que é posteriormente interpretado e executado por uma Java Virtual Machine (JVM). *O que é isso, professor?* É um programa que carrega e executa os aplicativos Java, convertendo bytecodes em código executável. *Lembram que eu falei que Java é uma Linguagem WORA?* Pois é, isso ocorre em grande parte por conta do bytecode e da Máquina Virtual Java.



Por conta deles, programas escritos em Java podem funcionar em qualquer plataforma de hardware e software que possua uma JVM, tornando assim essas aplicações independentes da plataforma, como apresenta a imagem acima (Win32, UNIX e MacOS)! Galera, qualquer plataforma... desde um computador a uma geladeira. A imagem abaixo é similar à anterior, apenas para solidificar!



Uma observação importante: Código Java é sempre compilado em um bytecode. No entanto, nem todo bytecode é proveniente de Código Java. *Como assim, professor?* É isso mesmo! Por exemplo: eu posso compilar o Código Ada em um *bytecode* e rodá-lo em uma JVM! *E quais outras linguagens?* Temos também Eiffel, Pascal, Python, C.

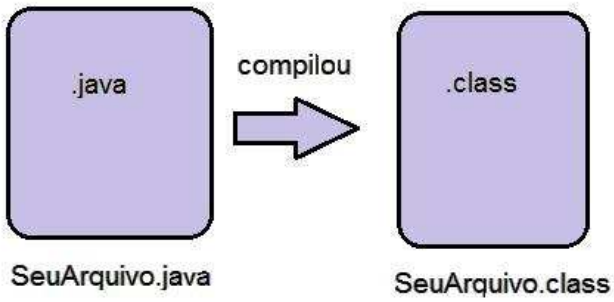
A JVM é capaz de entender bytecodes – assim como nós somos fluentes em português, ela é fluente em bytecode. Para criar bytecodes, basta seguir um conjunto de regras de formação. Logo, se existe um compilador que seja capaz de transformar o código-fonte (de qualquer linguagem) em uma bytecode seguindo as especificações corretamente, o bytecode poderá ser interpretado por uma JVM.

CURIOSIDADE

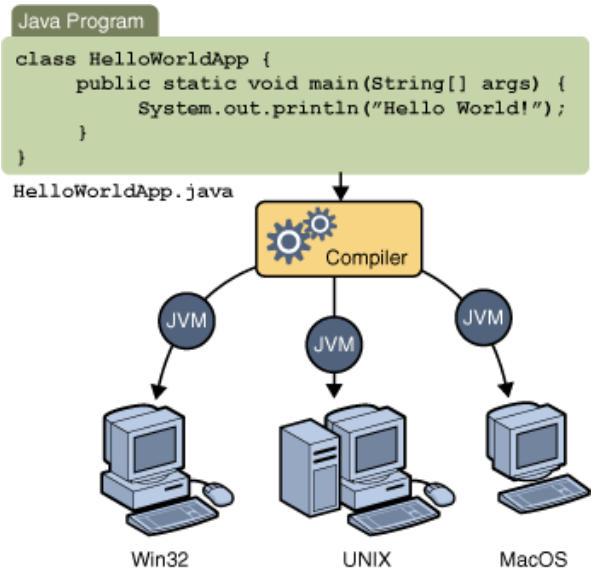
A Sun Microsystems declarou recentemente que existem atualmente cerca de 5.5 bilhões de dispositivos executando uma Java Virtual Machine (JVM).

1.3 JAVA: JAVA VIRTUAL MACHINE (JVM)

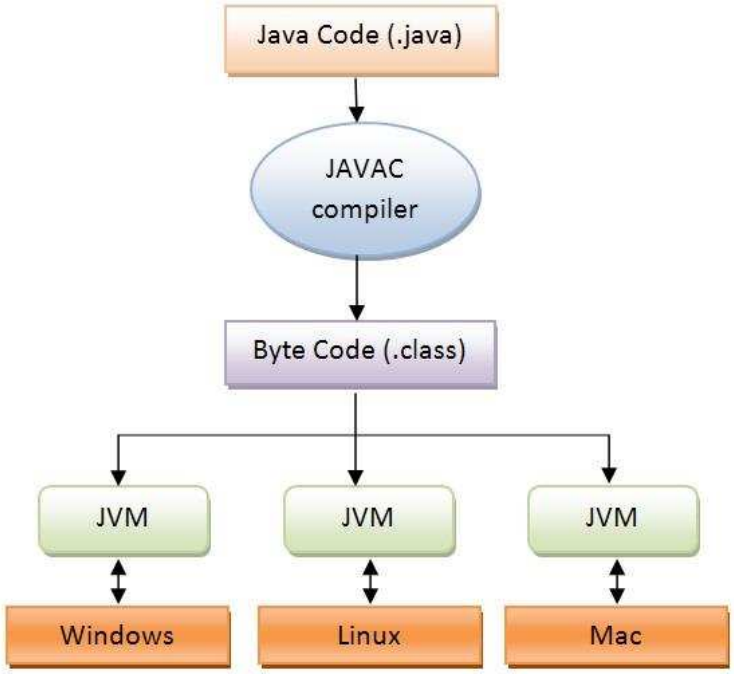
A Linguagem Java tem dois processos de execução de código-fonte: Compilação e Interpretação! Vamos lá... o programador escreve um código em Java em um editor de texto, por exemplo. Ele salva com a extensão .java e passa por um compilador (JavaC)! Esse compilador transforma o arquivo *.java* em código de máquina e em um arquivo *.class*, também chamado *bytecode* – como mostra a imagem abaixo.



O bytecode é um código intermediário, que é posteriormente interpretado e executado por uma Java Virtual Machine (JVM). *O que é isso, professor?* É um programa que carrega e executa os aplicativos Java, convertendo bytecodes em código executável. *Lembram que eu falei que Java é uma Linguagem WORA?* Pois é, isso ocorre em grande parte por conta do bytecode e da Máquina Virtual Java.



Por conta deles, programas escritos em Java podem funcionar em qualquer plataforma de hardware e software que possua uma JVM, tornando assim essas aplicações independentes da plataforma, como apresenta a imagem acima (Win32, UNIX e MacOS)! Galera, qualquer plataforma... desde um computador a uma geladeira. A imagem abaixo é similar à anterior, apenas para solidificar!



Uma observação importante: Código Java é sempre compilado em um *bytecode*. No entanto, nem todo *bytecode* é proveniente de Código Java. *Como assim, professor?* É isso mesmo! Por exemplo: eu posso compilar o Código Ada em um *bytecode* e rodá-lo em uma JVM! *E quais outras linguagens?* Temos também Eiffel, Pascal, Python, C.

A JVM é capaz de entender *bytecodes* – assim como nós somos fluentes em português, ela é fluente em *bytecode*. Para criar *bytecodes*, basta seguir um conjunto de regras de formação. Logo, se existe um compilador que seja capaz de transformar o código-fonte (de qualquer linguagem) em uma *bytecode* seguindo as especificações corretamente, o *bytecode* poderá ser interpretado por uma JVM.

CURIOSIDADE:

A Sun Microsystems declarou recentemente que existem atualmente cerca de 5.5 bilhões de dispositivos executando uma *Java Virtual Machine* (JVM).

Agora mudando um pouco de assunto, a JVM tem uma memória heap que é compartilhada entre todas as threads da máquina virtual. Essa pilha é a área de dados em tempo de execução a partir da qual a memória para todas as instâncias de classes e arrays é alocada. A pilha é criada durante a iniciação da máquina virtual java. *Até aqui, tudo bem?*

O armazenamento da heap para objetos é realizado por um sistema de gerenciamento de armazenamento automático (Garbage Collector), i.e., objetos nunca são desalocados explicitamente. A JVM não assume um tipo particular de sistema de gerenciamento de armazenamento automático e a técnica para tal pode ser escolhida de acordo com os requisitos do sistema do implementador.

A pilha pode ser de um tamanho fixo ou pode ser expandida de acordo com a necessidade de algum cálculo. Ela também pode ser contraída se uma pilha maior se tornar desnecessária. Se o cálculo necessitar de mais memória do que a pilha pode suportar, a JVM lança uma exceção `OutOfMemoryError`. Além disso, a memória para a pilha não precisa ser contígua.

Por fim, vamos falar um pouco sobre as ferramentas de monitoramento! O JConsole é uma ferramenta gráfica de monitoramento da máquina virtual e de aplicações Java para máquinas locais ou remotas. Ele utiliza recursos da máquina virtual para fornecer informações sobre o desempenho e o consumo de recursos de aplicativos em execução na plataforma usando JMX (Java Management Extensions).

O JPS (JVM Process Status) é outra ferramenta de monitoramento que lista hotspots da máquina virtual no sistema alvo. Outra importante função é listar os processos Java de um usuário. Por fim, o JStack imprime a pilha de rastros de um processo Java, arquivos ou servidores de depuração. Assim é possível rastrear eventuais erros e depurar o código.

1.4 JAVA: PASSAGEM POR VALOR E POR REFERÊNCIA

Vamos falar sobre passagem de parâmetros por valor e por referência. Vocês sabem que, quando o módulo principal chama uma função ou procedimento, ele passa alguns valores chamados Argumentos de Entrada. Esse negócio costuma confundir muita gente, portanto vou explicar por meio de um exemplo, utilizando a função DobraValor(valor1, valor2) – apresentada na imagem abaixo:

```
#include <stdio.h>

void DobraValor(int valor1, int valor2)
{
    valor1 = 2*valor1;
    valor2 = 2*valor2;

    printf("Valores dentro da Função: \nValor 1 = %d\n Valor 2 = %d\n",valor1,valor2);
}

int main()
{
    int valor1 = 5;
    int valor2 = 10;

    printf("Valores antes de chamar a Função:\nValor 1 = %d\nValor 2 = %d\n",valor1,valor2);
    DobraValor(valor1,valor2);
    printf("Valores depois de chamar a Função:\nValor 1 = %d\nValor 2 = %d\n",valor1,valor2);

    return();
}
```

Essa função recebe dois valores e simplesmente multiplica sua soma por dois. *Então o que acontece se eu passar os parâmetros por valor para a função?* Bem, ela receberá uma cópia das duas variáveis e, não, as variáveis originais. Logo, antes de a função ser chamada, os valores serão os valores iniciais: 5 e 10. Durante a chamada, ela multiplica os valores por dois, resultando em: 10 e 20.

```
Valores antes de chamar a Função:
Valor 1 = 5
Valor 2 = 10
Valores dentro da Função:
Valor 1 = 10
Valor 2 = 20
Valores depois de chamar a Função:
Valor 1 = 5
Valor 2 = 10
```

Após voltar para a função principal, os valores continuam sendo os valores iniciais: 5 e 10, como é apresentado acima na execução da função. Notem que os valores só se modificaram dentro da função DobraValor( ). *Por que, professor?* Ora, porque foi passada para função apenas uma cópia dos valores e eles que foram multiplicados por dois e, não, os valores originais.

```
#include <stdio.h>
void DobraValor(int *valor1, int *valor2)
{
        *valor1 = 2*(*valor1);
        *valor2 = 2*(*valor2);
        printf("Valores dentro da Função: \nValor 1 = %d\n Valor 2 = %d\n", *valor1, *valor2);
}
int main()
{
        int valor1 = 5;
        int valor2 = 10;

        printf("Valores antes de chamar a Função:\nValor 1 = %d\nValor 2 = %d\n",valor1,valor2);
        DobraValor(&valor1,&valor2);
        printf("Valores depois de chamar a Função:\nValor 1 = %d\nValor 2 = %d\n",valor1,valor2);

        return();
}
```

Professor, o que ocorre na passagem por referência? Bem, ela receberá uma referência para as duas variáveis originais e, não, cópias. Portanto, antes de a função ser chamada, os valores serão os valores iniciais: 5 e 10. Durante a chamada, ela multiplica os valores por dois, resultando em: 10 e 20. Após voltar para a função principal, os valores serão os valores modificados: 10 e 20.

```
Valores antes de chamar a Função:
Valor 1 = 5
Valor 2 = 10
Valores dentro da Função:
Valor 1 = 10
Valor 2 = 20
Valores depois de chamar a Função:
Valor 1 = 10
Valor 2 = 20
```

Notem que os valores se modificaram não só dentro da função DobraValor( ), como fora também (na função principal). *Por que isso ocorreu, professor?* Ora, porque foi passada para função uma referência para os valores originais e eles foram multiplicados por dois, voltando à função principal com os valores dobrados! Por isso, os valores 10 e 20.

Resumindo: a passagem de parâmetro por valor recebe uma cópia da variável original e qualquer alteração não refletirá no módulo principal. A passagem de parâmetro por referência recebe uma referência para a própria variável e qualquer alteração refletirá no módulo principal. Agora atenção máxima: em Java, a passagem de parâmetros é sempre, sempre, sempre por valor! *Bacana?*

1.5 JAVA: EMPACOTAMENTO

Utilizamos pacotes para organizar as classes semelhantes! Grosso modo, pacotes são apenas pastas ou diretórios do sistema operacional onde ficam armazenados os arquivos fonte de Java e são essenciais para o conceito de encapsulamento, no qual são dados níveis de acesso às classes. O empacotamento gera um arquivo .jar, que pode ser adicionado no *classpath* de uma aplicação.

Uma vez criada uma pasta (que será um pacote), deve-se definir para as classes a qual pacote elas pertencem. Isso é feito pela palavra-reservada **package**:

```
package br.com.site.pacoteEstrategia;
```

Essa deve ser a primeira linha de comando a ser compilada na classe. Java possui vários pacotes com outros pacotes internos e várias classes já prontas para serem utilizadas. Dentre os pacotes, podemos determinar dois grandes: o pacote **java**, que possui as classes padrões para o funcionamento da linguagem; e o pacote **javax**, que possui pacotes de extensão que fornecem ainda mais classes e objetos.

Em geral, aquelas classes fortemente conectadas às funções nativas do sistema operacional pertencem ao pacote **java** e as que não são tão fortemente conectadas pertencem ao pacote **javax**. Para utilizar os milhares de classes contidas nos inúmeros pacotes de Java devemos nos referenciar diretamente a classe ou importá-la. Para importar recursos de um pacote usamos a palavra-reservada **import**.

```
import javax.swing.JOptionPane;
```

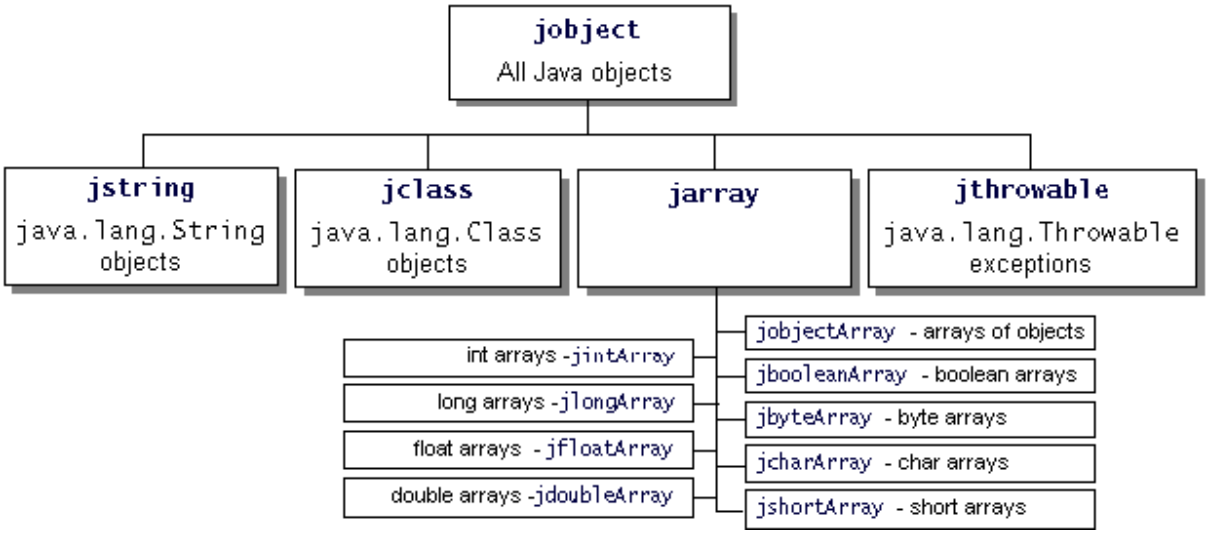
Para separar um pacote de seu sub-pacote, usam-se pontos (**br.com.site.pacoteEstrategia**). Ao utilizar o asterisco (*), todos os recursos do pacote são importados: **import java.awt.*** - isso importa todos os recursos pertencentes ao pacote **java.awt**. No entanto, podemos definir diretamente o pacote desejado: **import javax.swing.text.***. Isso irá importar apenas o sub-pacote **text** do pacote **javax.swing**.

A diferença entre as duas formas de importação de pacotes é o consumo de recursos do computador. Como o asterisco importa todos os recursos (classes, enumerações, interfaces), o consumo de memória será alto e, muito provavelmente, não usaremos todas as classes de todos os pacotes importados. Por isso, o recomendado é sempre importar apenas o pacote que será utilizado. A ordem é **Package**, **Import** e **Class**.

1.6 JAVA: RAÍZ

Muitas linguagens orientadas a objetos (como o C++) não suportam a noção de existir uma única classe a partir da qual todas as outras classes são derivadas, sem que isso seja um impedimento à linguagem. Entretanto, na linguagem Java, a falta desse tipo de classe tornaria a linguagem limitada. Todos os objetos da linguagem Java são de múltiplas camadas.

Cada classe, na hierarquia de classes, representa uma camada que adiciona diversas capacidades a um objeto. No topo desta hierarquia você sempre vai encontrar uma classe chamada de **Object** (Objeto). Qualquer classe estende implicitamente (sem necessidade de declarar) a classe **Object**. Claro que, na maioria das vezes, isso ocorre indiretamente. *Bacana, pessoal?*



1.7 JAVA: APPLET

Em 1995, a tecnologia Java e seu modelo de código móvel conhecido como applet foram criados pela Sun Microsystems. Applets são pequenos programas Java que são instalados em Contêineres Web e referenciados através de Páginas HTML em um navegador. Quando um browser acessa esta página HTML que referencia o applet, ele automaticamente também transfere o código do applet e o executa.

Applets, ao contrário das Servlets, rodam no Cliente e são hoje as principais responsáveis pela grande disseminação do conceito e das tecnologias de código móvel em geral. O uso de código móvel na web, e em especial o modelo de applets Java, permite que, adicionalmente ao uso de páginas HTML e imagens, os recursos transferidos entre servidores e clientes web sejam programas de computador.

A execução segura de código móvel se refere à relativa garantia de que o código oriundo de um servidor qualquer da web (chamado de código estrangeiro) não tenha acesso indiscriminado aos recursos presentes na plataforma do cliente e, portanto, não possa causar danos à plataforma do cliente, como ocorre com vírus e outros malwares. *Entendido?*

Dentro do contexto de uma aplicação web, normalmente, applets são utilizadas para se executar alguma validação, extração de dados ou operação do lado do dispositivo do cliente que o servidor não poderia realizar. Em uma aplicação comum, o ponto de partida para a sua execução é o método main. Em uma applet, porém, acontece de uma maneira um pouco diferente.

O primeiro código que uma applet executa é o código definido para sua inicialização, e seu construtor, através do método init que executa a inicialização básica da applet – geralmente usado para iniciar as variáveis globais, obter recursos de rede e configurar a interface. O método init é chamado apenas uma vez, quando o código da applet é carregado pela primeira vez, ao iniciar.

Então, o browser chama o método start a fim de informar que essa applet deve iniciar a sua execução. Se o usuário sair da página que contém o applet, o browser chama o método stop. No caso da finalização de uma thread, o método stop é acionado automaticamente. O método destroy é chamado para informar a applet que termine a sua execução, liberando espaço em memória.

Professor, posso ver um exemplo de applet? Claro! O exemplo mais clássico é aquele tecladinho digital que aparece para os correntistas digitarem sua senha no internet

banking. Abaixo eu mostro o meu caso: quando eu acesso o website da Claro para verificar minha conta de celular, eu preciso digitar minha senha por meio do tecladinho ao lado.

Galera, esse tecladinho está rodando no meu navegador! *Legal, não?* Pois é, applet não tem muito segredo! Applets Java estendem a classe java.applet.Applet. Vejamos o exemplo a seguir: ele ilustra o uso de Applets Java junto com o pacote AWT para criação da interface gráfica da applet. O intuito dessa applet é escrever na tela do navegador do usuário: Hello, World.

```
import java.applet.Applet;
import java.awt.*;

public class HelloWorld extends Applet
{
    public void paint(Graphics g) {
            g.drawString("Hello, world!", 20,10);
            g.drawArc(40, 30, 20, 20, 0, 360);
    }
}
```

2 JAVA: SINTAXE

2.1 JAVA: IDENTIFICADORES

Antes de tudo, é importante ressaltar que o Java é *case-sensitive* (Flamengo é diferente de flamengo). Identificador é o nome utilizado para representar variáveis, classes, objetos, métodos, pacotes, interfaces, etc. Por exemplo: na matemática, utiliza-se um nome para as incógnitas (x, y, z) que é o identificador daquela incógnita. No Java, existem um conjunto de regras para criação do identificador:

Deve ser a combinação de uma ou mais letras e dígitos UNICODE-16: Letras: A-Z; Letras: a-z; Underscore: _ ; Cifrão: $;Números: 0-9.

1. Não pode ser uma palavra-reservada (palavra-chave);
2. Não pode ser true, false ou null;
3. Não pode começar com números;
4. Não pode conter espaços em branco ou caracteres de formatação;

```
//CORRETO
int MyVariable, myvariable, MYVARIABLE;
int x, i, αρετη, OReilly;
int _myvariable, $myvariaBle, _9pins;

//INCORRETO
int My Variable;              //Contém espaço
int 9pins;                    //Começa com um dígito
int a+c, test-1, o'reilly&;   //Contém caractere não-alfanumérico
```

2.2 JAVA: BLOCOS E COMANDOS

Blocos de programação são aglomerados de instruções e declarações que têm escopo conjunto. Em outras palavras, as variáveis definidas como locais dentro de um bloco somente serão presentes dentro deste bloco, assim como as instruções ali presentes. Os blocos de programação são delimitados por chaves { } e podem ser aninhados, já os comandos sempre são terminados com ponto-e-vírgula.

```
import java.util.*;
import java.lang.*;
import java.io.*;

class Ideone {
   public static void main (String[] args) throws java.lang.Exception
   { //Início Bloco 1

      int a = 10;
      int b = 1;

      if (b==3)
      { //Início Bloco 2
         b = a*10;
      }
      else
      { //Início Bloco 3
         int a = 100;
         b = a*10;
      }
      System.out.println("O valor de b é " + b);

   }
}
```

2.4 JAVA: COMENTÁRIOS

Os comentários, como o próprio nome preconiza, são notas que podem ser incluídas no código-fonte de um programa para descrever o que o desenvolvedor desejar. Dessa forma, eles não modificam o programa executado e servem somente para ajudar o programador a melhor organizar os seus códigos. Os comentários em Java seguem a mesma sintaxe da linguagem C++:

```
import java.util.*;
import java.lang.*;
import java.io.*;

class Ideone {
   public static void main (String[] args) throws java.lang.Exception {

      /**
      * Comentário de mais de uma linha!
      **/

      int a = 10;
      int b = (int)(a*3.141592); //Comentário de única linha!

      System.out.println("O valor de b é " + b);
   }
}
```

2.5 JAVA: PALAVRAS RESERVADAS

Java possui 52 palavras-reservadas: três palavras para modificar acesso; treze palavras para modificar classes, variáveis e métodos; doze palavras para controle de fluxo; seis palavras para tratar erros; duas palavras para controlar pacotes; oito palavras para tipos primitivos; duas palavras para variáveis de referência; uma palavra para retorno de método; e duas palavras reservadas não utilizadas.

Espera, professor! Mas eu contei e deu 49 palavras! É que **null**, **true** e **false** não são consideradas tecnicamente palavras-reservadas, mas valores literais. No entanto, isso é só tecnicamente, porque caso se tente criar identificadores com essas palavras, resultará em erro de compilação. *Vamos ver todas as palavras reservadas?* A tabela abaixo apresenta cada uma e sua descrição:

Palavras	Descrição
abstract	Aplicado a um método ou classe indica que a implementação completa deste método ou classe é efetuada posteriormente, por uma subclasse. Caso seja uma classe, significa que ela não pode ser instanciada.
boolean	É um tipo de dados cujos valores podem ser true ou false.
break	Comando para controle de laço, no estilo C/C++.
byte	Tipo de dados inteiros com sinal, armazenado em formato binário na notação de complemento a dois e tamanho de 8 bits.
case	Indica uma opção entre várias em blocos switch.
catch	É utilizado juntamente com try, seu bloco é executado somente em caso de o programa lançar uma exceção do tipo indicado no seu parâmetro.
char	Para variáveis de caracteres, onde a sua representação interna equivale a um tipo numérico.
class	Para definir o início de um arquivo Java, todas as classes possuem pelo menos essa palavra-chave.

const	Essa palavra não tem uso específico em Java mas mesmo assim é uma palavra-chave.
continue	Para pular a iteração atual de uma estrutura de repetição.
default	Normalmente utilizado para o final de uma ou mais opções case´s de um bloco catch.
do	Estrutura de repetição que garante que o bloco será executado pelo menos uma vez durante a execução do programa.
double	Para variáveis numéricas e de pontos flutuantes com precisão de 64 bits.
else	Complemento de estrutura de condição.
enum	Palavra-chave adicionada na versão 5 do Java; é um tipo específico de dados que assemelha-se com uma classe que tem operações e dados internos.
extends	Utilizado para aplicar o conceito de herança para uma classe, onde uma classe receberá os métodos e variáveis de instância da classe chamada de pai.
final	Marca uma variável, classe ou método para que não seja possível modificar o seu valor ou comportamento no decorrer da execução do programa.
finally	Compõe o início de um bloco que sempre é executado para um bloco de tratamento de erros, mais utilizado para limpar recursos que foram abertos no bloco de tratamento.
float	Variáveis numéricas e de pontos flutuantes com precisão de 32 bits.
for	Estrutura de repetição que declara, testa e incrementa variável para uso local.
goto	Não tem uso específico na linguagem.

if	Estrutura de condição mais comum na linguagem.
implements	Informa que uma determinada classe irá implementar uma determinada interface.
import	Para relacionar classes externas à atual, permitindo o uso de nomes mais curtos para recursos da classe externa.
instanceof	Testa se um objeto é uma instância de uma classe específica ou se é null.
int	Para variáveis numéricas de precisão -2.147.483.648 até 2.147.483.647.
interface	Informa que o modelo não é uma classe, mas sim um protótipo de classe sem implementação para os métodos, obrigando as classes que a implementarão a seguir as determinadas regras.
long	Para variáveis numéricas de precisão de 64 bits.
native	Métodos marcados como native dizem que sua implementação é feita em uma outra linguagem (por exemplo, C), para que se possa acessar recursos específicos do sistema operacional.
new	Utilizada para se criar novas instâncias de objetos.
package	Informa em que estrutura de diretórios a classe está localizada.
private	Marca a visibilidade de um método ou variável de instância para que apenas a própria classe acesse.
protected	Marca a visibilidade de um método ou variável de instância para que a própria classe ou suas filhas acessem.

public	Marca a visibilidade de uma classe, método ou variável de instância para que todas as classes em todos os pacotes tenham acesso.
return	Devolve para o método chamador de um valor que é do mesmo tipo declarado na assinatura do método.
short	Para variáveis numéricas de precisão de -32.768 até 32.767.
static	Marca um método ou variável para que se tenha apenas uma cópia da memória desse membro.
strictfp	Serve para aumentar a precisão em operações com pontos flutuantes.
super	Chama membros da classe-pai.
switch	Representa blocos de decisões de fluxos semelhantes ao if, mas com mais organização em determinadas situações.
synchronized	Um método com essa marcação será controlado para que não se possa ter duas threads acessando o mesmo objeto.
this	Representa a instância que está atualmente sendo executada.
throw	É utilizado para lançar uma exceção.
throws	É utilizado para se declarar que um método pode lançar uma exceção.
transient	Indica que uma determinada variável de instância não será serializada junto com o objeto da classe.
try	Para executar métodos que têm chances de lançar exceções, mas que serão tratados em blocos catch que o seguirão.

void	Representa um retorno vazio, i.e., nenhum retorno para esse método.
volatile	Indica que uma determinada variável de instância pode ser modificada em duas threads distintas ao mesmo tempo.
while	Bloco de repetição que será executado enquanto seu parâmetro estiver retornando true.

2.7 JAVA: TIPOS PRIMITIVOS

A linguagem Java oferece um total de oito tipos primitivos para criação de programas. Esses tipos são utilizados para declarar variáveis que auxiliam na construção dos algoritmos. Apesar de a linguagem oferecer tantos tipos, muitos deles são capazes de representar os mesmos tipos de dados, mas com uma capacidade de armazenamento maior (maior quantidade de bits).

Variáveis do tipo **byte**, **short**, **int** e **long**, por exemplo, podem ser usadas para representar números inteiros, variando em diferentes faixas de valores, embora os tipos **int** e **long** sejam os mais utilizados. O mesmo acontece com variáveis do tipo **float** e **double**, que são usadas para representar números reais. O tipo **boolean** é usado para declarar variáveis que podem assumir um os valores: **true** ou **false**.

NOME	TIPO	TAMANHO	MÍNIMO	MÁXIMO	DEFAULT
LÓGICO	boolean	-	false	true	false
CARACTERE	char	16 bits	0	$2^{15} - 1$	'\u0000'
INTEIRO	byte	8 bits	-2^{7}	$2^{7} - 1$	0
	short	16 bits	-2^{15}	$2^{15} - 1$	0
	int	32 bits	-2^{31}	$2^{31} - 1$	0
	long	64 bits	-2^{63}	$2^{63} - 1$	0
DECIMAL	float	32 bits	7 Casas Decimais		0.0
	double	64 bits	15 Casas Decimais		0.0

Variáveis booleanas são bastante usadas em comandos condicionais ou de repetição. Finalmente, o tipo **char** é utilizado para representar caracteres, como, por exemplo, as letras de alfabetos de línguas de diferentes países. Apesar de, em alguns programas, ser necessário o uso do tipo **char** para representar e permitir o processamento de caracteres de uma frase, o mais comum é usar a classe **String**.

Galera, vou enfatizar isso novamente porque é realmente importante! **String** não é um Tipo Primitivo! *Bacana?* **String** não é um Tipo Primitivo, nunca foi e jamais será um Tipo Primitivo. Além disso, a Classe **String** pertence ao pacote **Lang** e herda diretamente de **Object**. Outra coisa: posso escrever números na forma decimal (**8**), octal (**08**) ou hexadecimal (**0x8**).

OBSERVAÇÕES

O valor-padrão para um número decimal é Double; se o programador quiser que seja Float, deve enviar como parâmetro o valor acrescido da letra f (Ex: 3.14f). Da mesma forma, o valor-padrão para inteiro é um int. O único tipo primitivo que não pode ser atribuído a nenhum outro tipo é o boolean.

2.8 JAVA: OPERADORES

Os operadores são sinais que representam atribuições, cálculos e ordem dos dados. As operações seguem uma ordem de prioridades, ou seja, alguns cálculos são processados antes de outros, assim como ocorre na matemática. Para manipular os valores das variáveis de um programa, devemos utilizar os operadores oferecidos pela linguagem de programação adotada.

ARITMÉTICOS	ATRIBUIÇÃO	RELACIONAIS	LÓGICOS	BIT A BIT
+	=	>	!	&
-	+=	<	&&	|
*	-=	>=	||	^
/	*=	<=		<<
%	/=	!=		>>
	%=	==		>>>
	++	?		
	--	instanceof		

Operadores Aritméticos: +, -, *, /, %.

```
$Numero = 2 + 8;       //$Numero = 2 + 8 = 10
$Numero = 8 - 2;       //$Numero = 8 - 2 = 6
$Numero = 2 * 8;       //$Numero = 2 * 8 = 16
$Numero = 8 / 2;       //$Numero = 8 / 2 = 4
$Numero = 2 % 8;       //$Numero = 2 % 8 = 2 (Resto da divisão de 2 por 8)

String $palavra1 = "Alô";
String $palavra2 = "Mundo!";

String $frase = $palavra1 + ", " + $palavra2; //$frase = "Alô, Mundo!"
```

Operadores de Atribuição: =, +=, -=, *=, /=, %=, ++, --.

```
int $Numero = 5;    //Atribui-se o valor 5 a variável $Numero

$Numero ++;         //$Numero = $Numero + 1 = 5 + 1 = 6
$Numero --;         //$Numero = $Numero - 1 = 6 – 1 = 5

$Numero += 3;       //$Numero = $Numero + 3 = 5 + 3 = 8
$Numero -= 3;       //$Numero = $Numero - 3 = 8 – 3 = 5

$Numero *= 3;       //$Numero = $Numero * 3 = 5 * 3 = 15
$Numero /= 3;       //$Numero = $Numero / 3 = 15 / 3 = 5

$Numero %= 3;       //$Numero = $Numero % 3 = 5 % 3 = 2 (Resto da divisão de 5 por 3)
```

Operadores Relacionais: >, <, >=, <=, ==, != , ? , instanceof.

```
boolean Variavel_1;

Variavel_1 = 4  <  4;  //FALSE:  4 não é menor que 4
Variavel_1 = 4  <= 4;  //TRUE:   4 é menor ou igual a 4
Variavel_1 = 3  >  7;  //FALSE:  3 não é maior que 7
Variavel_1 = 3  >= 7;  //FALSE:  3 não é maior ou igual a 7
Variavel_1 = 1  == 1;  //TRUE:   1 é igual a 1
Variavel_1 = 2  != 1;  //TRUE:   2 é diferente de 1

int Variavel_2 = 4;
int Variavel_3 = 8;

Variavel_2 = (Variavel_2 > Variavel_3)? Variavel_2: Variavel_3;
//Variavel_2: 4 não é maior que 8, logo Variavel_2 recebe o valor de Variavel_3
Variavel_3 = (Variavel_2 < Variavel_3)? Variavel_2: Variavel_3;
//Variavel_3: 4 é menor que 8, logo Variavel_3 recebe o valor de Variavel_2

Moto  $cg500 = new Moto();
Carro $celta = new Carro();

boolean Teste = $celta instanceof Carro; //TRUE:  $celta é uma instância de Carro
boolean Teste = $cg500 instanceof Carro; //FALSE: $cg500 não é uma instância de Carro
```

Vamos ver abaixo um exemplo mais complexo:

```
//Caso 1: y = 2 e x = 2
int x = 0;
int y = x++ + ++x;

//Caso 2: y = 1 e x = 2
int x = 0;
int y = x++ + x++;

//Caso 3: y = 3 e x = 2
int x = 0;
int y = ++x + ++x;

//Caso 4: y = 2 e x = 2
int x = 0;
int y = ++x + x++;
```

Operadores Lógicos: !, &&, ||.

```
boolean $Variavel;

$Variavel = (2<45) && (45<2) //$Variavel = TRUE && FALSE = FALSE
$Variavel = (2<45) || (45<2) //$Variavel = TRUE || FALSE = TRUE

!$Variavel //$Variavel = FALSE;
!$Variavel //$Variavel = TRUE;
```

Operadores Bit a Bit: &, |, ^, <<, >>, >>>.

```
int $Variavel;
$Variavel = 34 & 435;  //000100010 & 110110011 = 100010 = 34 (Operação AND)
```

```
$Variavel = 34^46;    //000100010 ^ 000101110 = 000001100 = 12 (Operação XOR)
$Variavel = 436|547;  //0110110100 | 1000100011 = 1110110111 = 951 (Operação OR)

int $Variavel = -3;    //$Variavel vale -3
$Variavel = $Variavel >>  1   //$Variavel = 11111101 >> 1 = 11111110 = -2 (SHIFT RIGHT)
$Variavel = $Variavel <<  1;  //$Variavel = 11111110 << 1 = 11111100 = -4 (SHIFT LEFT)
$Variavel = $Variavel >>> 1;  //$Variavel = 11111100 >>> 1 = NÚMERO GIGANTE
```

2.9 JAVA: VETORES

Um vetor é uma estrutura de dados formada por um conjunto de dados ou outros elementos de um mesmo tipo, podendo ter uma dimensão ou mais (quando tem duas, é chamado de matriz) e cujo acesso aos dados é feito através de índices. Cada item de um vetor é chamado de elemento. Cada um dos elementos possui uma posição dentro do vetor, à qual referenciamos através do índice do elemento.

Para declarar um vetor e de uma matriz, devemos utilizar a seguinte sintaxe:

```
//Declaração 1
tipo[] identificador1;
tipo identificador1[];

//Declaração 2
tipo[] identificador2 = new tipo[];
tipo identificador2[] = new tipo[];

//Declaração 3 (Para duas dimensões)
tipo[][] identificador3;
tipo identificador3[][];
```

Para inicializar um vetor, devemos utilizar a seguinte sintaxe:

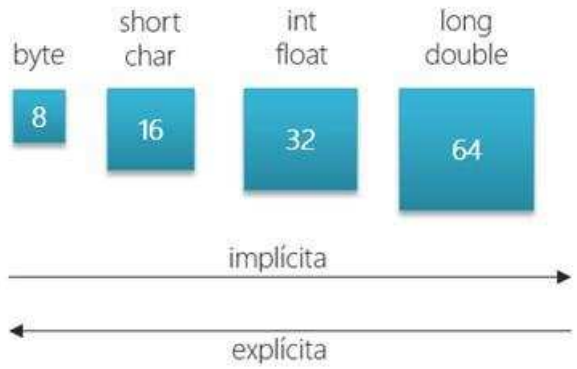
```
//Inicialização 1
int[] Vetor1 = {34,27,91,56};

//Inicialização 2 (Inválida)
int[] Vetor2 = new int[4];
Vetor2 = {34,27,91,56};

//Inicialização 3
int[] Vetor3 = new int[4];
Vetor3[0] = 34;
Vetor3[1] = 27;
Vetor3[2] = 91;
Vetor3[3] = 56;
```

2.11 JAVA: CONVERSÃO DE TIPOS

Muitas vezes precisamos fazer cálculos e guardar o resultado em alguma outra variável para uso posterior, porém o tipo de resultado pode não condizer com o tipo da variável que irá receber esse resultado, seja de outra variável ou resultado de uma expressão matemática. A conversão de tipos primitivos é a transformação de um tipo para outro. Essas conversões podem ser implícitas ou explícitas.



As conversões implícitas ocorrem quando atribuímos um valor de menor tipo em uma variável de tipo maior. Este tipo de conversão também é conhecido como conversão de ampliação e ocorrerá de forma automática porque um valor de menor tipo sempre caberá em uma variável de maior tipo. Podemos ver um exemplo representado abaixo:

```
import java.util.*; import java.lang.*; import java.io.*;

class Ideone {
   public static void main (String[] args) throws java.lang.Exception {

      System.out.println("Conversão Implícita:");
      double a1 = 10*5.2 + 4 - 1.3;
      System.out.println("Variável a1 = " + a1);
      double a2 = 5/2;
      System.out.println("Variável a2 = " + a2);
      double a3 = 5/2.0;
      System.out.println("Variável a3 = " + a3);

      System.out.println("\nConversão Explícita:");
      int a4 = (int) (10*5.2 + 4 - 1.3);
      System.out.println("Variável a4 = " + a4);
      int a5 = (int) (5/2.0);
      System.out.println("Variável a5 = " + a5);
   }
}
```

O resultado das conversões implícitas e explícitas são apresentados abaixo:

```
Conversão Implícita:
```

```
Variável a1 = 54.7
Variável a2 = 2.0
Variável a3 = 2.5

Conversão Explícita:
Variável a4 = 54
Variável a5 = 2
```

A conversão explícita ou de redução ocorre quando precisamos atribuir um valor de tipo maior para uma variável de tipo menor. Nesse caso, precisamos informar diretamente entre parênteses (antes da variável, literal ou resultado de uma expressão) o tipo de dado que vamos converter. Existem 19 conversões implícitas e 22 conversões explícitas possíveis.

2.12 JAVA: CONTROLE DE FLUXO

Professor, o que é Controle de Fluxo? É como se controla o fluxo de um código, isto é, como um programa executa suas tarefas! Por meio de comandos, tarefas podem ser executadas seletivamente, repetidamente ou excepcionalmente. Não fosse o controle de fluxo, um programa poderia executar apenas uma única sequência de tarefas, perdendo seu dinamismo.

Em Java, temos duas estruturas: Seleção e Repetição. A Estrutura de Seleção (ou Condição) consiste basicamente dos comandos **if-else**. Ele é empregado para executar seletivamente ou condicionalmente um outro comando mediante um critério de seleção. Esse critério é dado por uma expressão, cujo valor resultante deve ser um dado do tipo booleano, isto é, **true** ou **false**.

Se esse valor for **true**, então o outro comando é executado; se for **false**, a execução do programa segue adiante. A sintaxe do **if-else** é apresentada abaixo:

```
if(CondiçãoBooleana) {
   comando1;
   comando2;
   (...)
   comandoN;
}
 else {
   comando1;
   comando2;
   (...)
   comandoN;
}
```

Uma variação desse comando, o **if-else** permite escolher alternadamente entre dois outros comandos a executar. Nesse caso, se o valor da expressão condicional que define o critério de seleção for **true**, então o primeiro dos outros dois comandos é executado, do contrário, o segundo. Lembrando que é possível aninhar ou combinar vários comandos **else** e **if** – o último **else** é opcional.

```
int idade;

if (idade <= 1)
   System.out.println("Bebê");
else if(idade > 1 && idade <= 10)
   System.out.println("Criança");
else if(idade > 10 && idade <= 13)
   System.out.println("Pré-adolescente");
else if(idade > 13 && idade <= 18)
   System.out.println("Adolescente");
else
   System.out.println("Adulto");
```

Existem algumas situações em que se sabe de antemão que as condições assumem o valor **true** de forma mutuamente exclusiva, i.e., apenas uma entre as condições sendo testadas assume o valor **true** ao mesmo momento. Nesses casos, a linguagem Java provê um comando de controle de fluxo bastante poderoso. Trata-se do comando **switch**, cuja sintaxe é a seguinte:

```
switch([expressão]) {
  case [constante 1]:
    [comando 1]
    break;
  case [constante 2]:
    [comando 2]
    break;
  ...
  case [constante n]:
    [de comando n]
    break;
  default:
    [comando]
}
```

A **[expressão]** pode ser qualquer expressão válida. Ela é avaliada e o seu valor resultante é comparado com as constantes distintas **[constante 1]**, **[constante 2]**, ..., **[constante n]**. Caso esse valor seja igual a uma dessas constantes, o respectivo comando é executado (e todos os demais são saltados). Se o valor for diferente, o comando presente sob o rótulo default: é executado. Vejamos outro exemplo:

```
int mesAtual = 5;

switch (mesAtual) {
   case 1:
      System.out.println("Janeiro"); break;
   case 2:
      System.out.println("Fevereiro"); break;
   case 3:
      System.out.println("Março"); break;
   case 4:
      System.out.println("Abril"); break;
   case 5:
      System.out.println("Maio"); break;
   case 6:
      System.out.println("Junho"); break;
   case 7:
      System.out.println("Julho"); break;
   case 8:
      System.out.println("Agosto"); break;
   case 9:
      System.out.println("Setembro"); break;
   case 10:
      System.out.println("Outubro"); break;
   case 11:
      System.out.println("Novembro"); break;
   case 12:
      System.out.println("Dezembro"); break;
```

```
    default:
            System.out.println("Mês inválido.");
}
```

Chegamos, então, à Estrutura de Repetição (ou Iteração)! Frequentemente, desejamos que uma tarefa seja executada repetidamente por um programa enquanto uma dada condição seja verdadeira. Isso é possível pela utilização do comando **while**. Este comando avalia uma expressão condicional, que deve resultar no valor **true** ou **false**.

Se o valor for **true**, então o comando subjacente é executado; se a expressão for **false**, então o comando é saltado e a execução prossegue adiante. A diferença é que após executar o comando subjacente, a expressão condicional é novamente avaliada e seu resultado novamente considerado. Desse modo a execução do comando subjacente se repetirá até que o valor da expressão condicional seja **false**.

Observe, porém, que a expressão é avaliada antes de uma possível execução do comando subjacente, o que significa que esse comando pode jamais ser executado. O comando **while** é portanto pré-testado, isto é, antes de executar qualquer comando, testa-se a condição oferecida. Caso seja verdadeira, realiza os comandos internos; caso seja falsa, sequer realiza qualquer comando. A sintaxe é:

```
while ([condição])
  [comando subjacente]
```

Deve-se ter cuidado para não acabar implementando um laço infinito (desde que essa não seja sua intenção). Um laço infinito é um laço em que a condição de saída nunca é satisfeita, portanto ele roda eternamente. Uma variação do comando **while** que funciona de maneira bastante análoga é o **do-while**. A diferença é que ele é pós-testado, isto é, executa os comandos internos e só depois avalia a condição.

```
do
  [comando]
while ([condição]);
```

Em certas situações, precisamos de laços de repetições nos quais alguma variável é usada para contar o número de iterações. Para essa finalidade, temos o laço for. Este é o tipo de laço mais geral e mais complicado disponível na linguagem Java. Esse laço é pré-testado ou pós-testado? Fácil, é pré-testado! Avalia-se a condição antes de executar os comandos. Sua sintaxe é a seguinte:

```
for ([expressão 1]; [condição]; [expressão 2])
  [comando]
```

A [expressão 1] é chamada expressão de inicialização, [condição] é uma expressão condicional e [expressão 2] é uma expressão qualquer a ser executado no final de cada iteração. O laço **for** avalia inicialmente a expressão de inicialização. Em seguida, avalia a expressão condicional. Se o valor desta for **true**, então o comando é executado.

A segunda expressão é avaliada em seguida, e finalmente o laço volta a avaliar novamente a expressão condicional. Do contrário, se o valor da expressão for **false**, a execução prossegue adiante do laço for. Observem que é completamente possível transformar um **for** em um **while**. Podemos dizer que eles são equivalentes, mas escritos de maneira diferente.

```
[expressão 1]
while ([condição]) {
   [comando]
   [expressão 2]
}
```

Agora vamos ver um exemplo do **while**:

```
int idade = 26;

while (idade > 30) {
   System.out.println("Minha idade é " + idade);
   idade = idade + 1;
}
```

```
Resultado:

...
```

Agora vamos ver um exemplo do **do-while**:

```
int idade = 26;

do {
   System.out.println("Minha idade é " + idade);
   idade = idade + 1;
} while (idade > 30)
```

```
Resultado:

Minha idade é 26
```

Agora vamos ver um exemplo do **for**:

```
int idade;
```

```
for (idade = 26; idade < 30; idade++) {
   System.out.println("Minha idade é " + idade);
}
```

Resultado:

Minha idade é 26
Minha idade é 27
Minha idade é 28
Minha idade é 29

3 JAVA: ORIENTAÇÃO A OBJETOS

3.1 JAVA: CLASSES

A classe é a planta ou esquema que indica como os objetos são criados, quais os seus comportamentos e variáveis de estado. Para declarar uma classe, é necessário utilizar a sintaxe a seguir:

```
[palavra-chave] class NomeDaClasse
{
   //Atributos e Métodos
}
```

Portanto para declarar uma classe, deve-se colocar a palavra **class** seguida de um identificador que irá servir de nome para a classe. O identificador pode ser qualquer palavra, exceto palavras reservadas. Por exemplo: class *Conta* introduz a declaração de uma nova classe chamada **Conta**. Note que, por convenção, o nome de uma classe inicia sempre com uma letra maiúscula. A Palavra-Chave é opcional, podendo ser:

```
//Essa classe pode ser acessada por todos
public class Carro {...}

//Essa classe não pode gerar instâncias
abstract class Carro {...}

//Essa classe não pode ser estendida
final class Carro {...}
```

3.3 JAVA: OBJETOS

Um objeto é uma instância de uma classe. Para criar um objeto, devemos utilizar a seguinte sintaxe:

```
new construtor();
```

O comando new, também conhecido como operador de criação, cria um novo objeto, alocando memória para o objeto e inicializando essa memória para valores default. Ele necessita de um operando: o construtor, que é o nome de um método especial que constrói o objeto. Uma vez construído, o objeto deve ser atribuído a uma variável, para que possa ser utilizado e referenciado no futuro.

```
/* 1) Operador NEW é responsável por criar um objeto;
 * 2) NomeClasse() é o construtor da Classe NomeClasse;
 * 3) NomeObjeto é uma variável do Tipo NomeClasse; */

NomeClasse NomeObjeto = new NomeClasse();

/* Observem que é possível atribuir o objeto de uma
 * classe para uma variável de outra classe */
```

A linguagem Java assume a responsabilidade de destruir qualquer objeto criado que não esteja sendo usando. Para tal, utiliza um Coletor de Lixo (Garbage Collector), que é executado em intervalos regulares, examinando cada objeto para ver se ele ainda é referenciado por alguma variável. Caso o objeto não seja utilizado ao menos por uma variável, ele é destruído e sua memória é liberada.

3.4 JAVA: ATRIBUTOS

Um atributo ou campo é uma variável declarada no corpo de uma classe. Ele serve para armazenar o estado de um objeto (atributo de instância) ou o estado de uma classe (atributo de classe). A sintaxe de declaração de um atributo é a seguinte:

```
[palavra-chave] tipoAtributo NomeAtributo [=expressão];
```

A Palavra-Chave é opcional, podendo ser:

- Final, Volatile, Static ou Transient

```
class Empregado
{
   final String Nome;  //Indica que Nome é um atributo constante;
   volatile Salario;   //Indica que Salário é modificável por threads distintas;
   static Idade;      //Indica que Idade é compartilhada por todos objetos;
   transient Sexo;    //Indica que Sexo não pode ser serializável;
}
```

- Modificadores de Acesso:

```
class Empregado
{
   public String nome;     //Nome: público
   private int Salario;    //Salário: privado
   protected short Idade;  //Idade: protegido
   char Sexo;              //Sexo: default
}
```

3.5 JAVA: MÉTODOS

Java utiliza métodos para se referir a trechos de código que são associados a classes. Se os atributos servem para manter ou armazenar o estado ou o valor de um objeto, os métodos servem para descrever os comportamentos de um objeto ou classe. Um método é muito similar a uma função em C. A maior diferença é que os métodos da linguagem Java são declarados completamente dentro de uma classe.

```
[Palavras-Chave] TipoRetorno NomeMetodo ([Lista de Parâmetros])
{
   //Corpo do Método
}
```

A sintaxe de declaração de um método é apresentada acima. A Palavra-Chave é opcional, podendo ser:

- Abstract, Final, Static, Native e Synchronized

```
//Indica que esse método não possui corpo
abstract int soma (int a, int b) {//...//}

//Indica que esse método não pode ser sobrescrito
final int soma (int a, int b) {//...//}

//Indica que esse método só pode acessar atributos de classe e não pode ser sobrescrito
static int soma (int a, int b) {//...//}

//Indica que esse método foi escrito outra linguagem
native int soma (int a, int b) {//...//}

//Indica que esse método só é executável por uma thread por vez
synchronized soma (int a, int b) {//...//}
```

- Modificadores de Acesso:

```
public int soma (int a, int b) {//...//}
private int soma (int a, int b) {//...//}
protected int soma (int a, int b) {//...//}
int soma (int a, int b) {//...//}
```

Vamos falar agora sobre um método importante: Construtor! Ele é um método especial, chamado pelo operador *new* quando um novo objeto necessita ser criado. Dentro do construtor, pode-se colocar código customizado de inicialização do objeto. Em geral, ele deve ter o mesmo nome da classe em que for declarado. Além disso, um construtor não deve ter um tipo de retorno em sua declaração.

Professor, não entendo uma coisa! Em geral, no código da classe não há nenhum método com o nome do método construtor. Ora, não há declaração de método algum! Como é isso? É verdade! Basicamente, quando você não declara um construtor para uma classe, o compilador cria um construtor padrão com uma lista vazia de parâmetros. Pode-se criar diversos construtores para uma mesma classe.

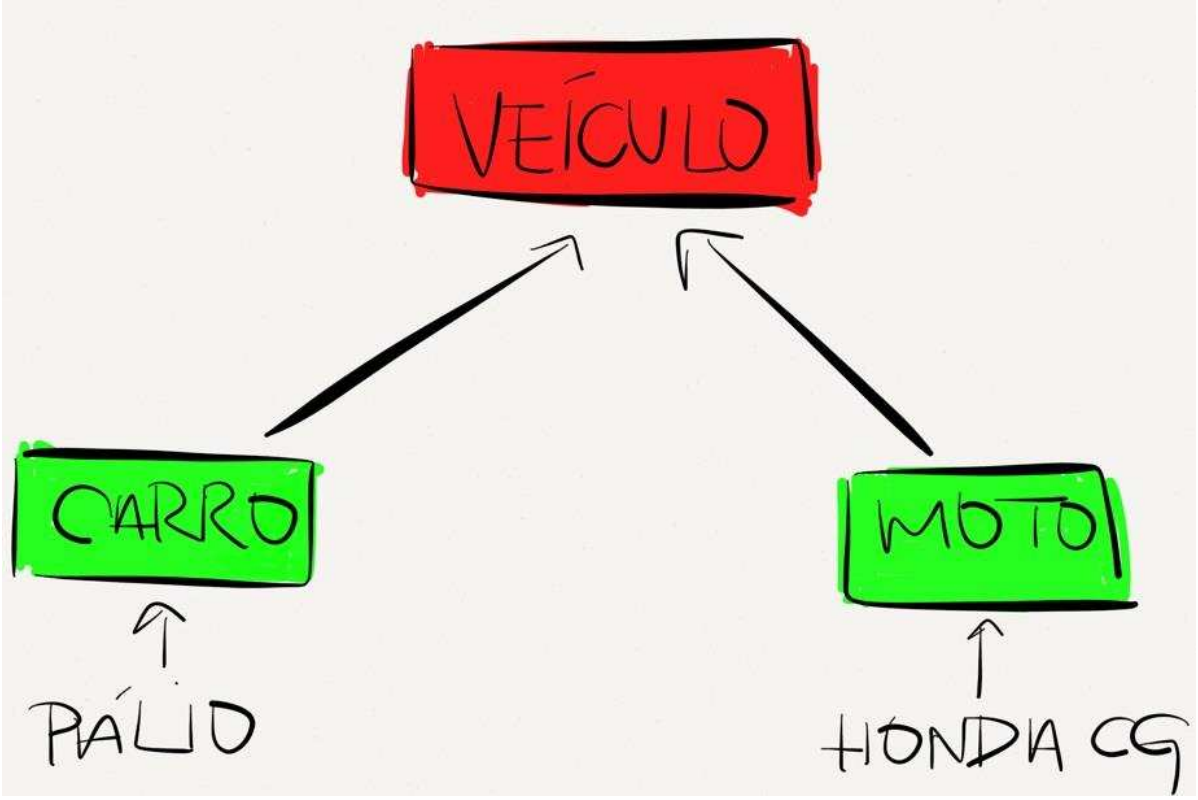
Outro método muito importante é o Método **Main**! Pois é, ele é sempre definido como um método de classe, i.e., possui um modificador **static**:

```
public static void main (String[] args) throws java.lang.Exception
```

3.6 JAVA: HERANÇA

Herança é a habilidade de se derivar alguma coisa específica a partir de algo mais genérico. Nós encontramos essa habilidade ou capacidade diversas vezes em nosso cotidiano. Por exemplo: um Pálio estacionado na garagem do seu vizinho é uma instância específica da categoria Carro, mais genérica. Da mesma forma, uma Honda CG 125 é uma instância específica da categoria mais genérica Moto.

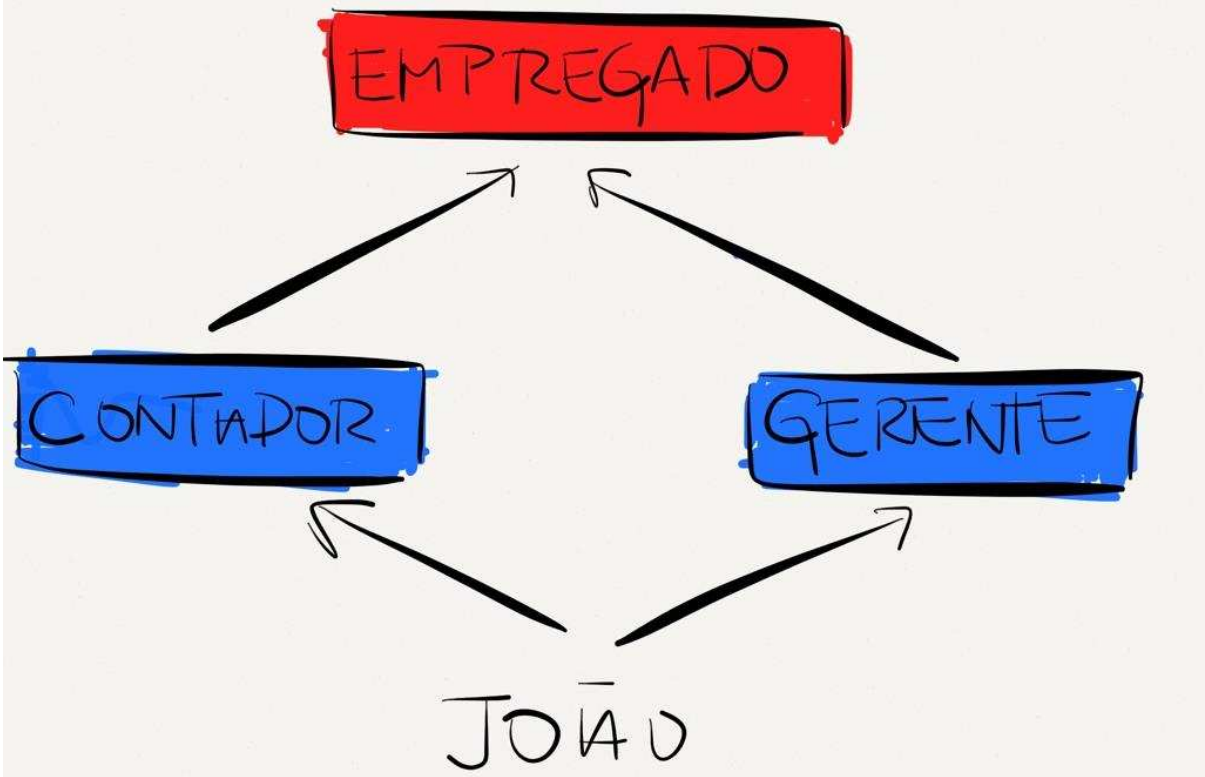
Se levarmos as categorias Carro e Moto para um outro nível mais elevado, as duas se relacionarão uma com a outra por serem instâncias específicas de uma categoria mais genérica ainda que elas: a categoria Veículo. Em outras palavras, carros e motos são veículos. A imagem abaixo esquematiza as relações entre Pálio, Carro, Honda CG 125, Moto e Veículo.



Esse exemplo ilustrou a Herança Simples! Neste caso, uma entidade herda estados e comportamentos (atributos e métodos) de uma e somente uma categoria. O Pálio, por exemplo, herda da categoria Carro (e somente dela, diretamente). Em contraste, a Herança Múltipla permite que uma entidade herde diretamente comportamentos e estados de duas ou mais categorias ao mesmo tempo.

Imagine um Empregado chamado José de uma empresa qualquer. Para ser mais específico, pense em José como sendo tanto Gerente quanto Contador

simultaneamente dessa empresa. *Ele herdaria as capacidades de um Gerente e de um Contador, concordam?* Portanto, em uma hierarquia de entidades, João herdaria de duas classes diferentes diretamente, como apresenta a imagem abaixo:



A sintaxe da herança sugere que você pode estender uma e somente uma classe. O Java não suporta Herança Múltipla, porque – segundo projetistas da linguagem – esse tipo de implementação poderia gerar confusão. Imagine, por exemplo, duas classes-base declarando um atributo que possua o mesmo nome mas com tipos diferentes. *Qual dos dois a classe-filha deveria herdar?*

A mesma situação poderia acontecer com um método! *Se dois métodos possuíssem o mesmo nome, mas diferentes listas de parâmetros ou tipos de retorno, qual deles a subclasse deveria herdar? Vocês percebem como isso poderia causar inconsistências de projeto?* Para prevenir tais problemas, a linguagem Java rejeita a implementação de herança múltipla.

Novas classes derivam capacidades (expressas como atributos e métodos) de classes já existentes. Isso faz com que o tempo de desenvolvimento de uma aplicação seja bem menor, pois classes já existentes e comprovadamente funcionais (livres de erros e já testadas) são reaproveitadas (ou reutilizadas). A sintaxe que expressa o conceito de extensão de classes é a seguinte:

```
class NomeClasseFilha extends NomeClassePai
{
  //Atributos e Métodos
}
```

Essa sintaxe pode ser lida da seguinte forma: NomeClasseFilha estende NomeClassePai. Em outras palavras, **NomeClasseFilha** herda (ou deriva) capacidades (expressas através de atributos e métodos) de **NomeClassePai**. A **NomeClasseFilha** é conhecida como subclasse, classe derivada ou classe-filha e **NomeClassePai** é conhecida como superclasse, classe-base ou classe-pai.

A palavra-chave **extends** faz com que uma subclasse herde (receba) todos os atributos e métodos declarados na classe-pai (desde que ela não seja final), incluindo todas as classes-pai da classe-pai. A classe-filha pode acessar todos os atributos e métodos não-privados. Ela herda, mas não acessa (ao menos diretamente) métodos e atributos privados.

Todo objeto criado a partir de uma subclasse é também um objeto do tipo da sua superclasse (Ex: um objeto do tipo Carro também é um objeto do tipo Veículo). Essa afirmação implica o fato de que você pode atribuir um objeto de uma subclasse para uma referência criada ou declarada para um objeto de sua superclasse. *Como assim, professor?* Vejamos!

```
//Objeto do tipo Carro é um objeto do tipo Veículo
Veiculo v = new Carro();
```

A linha de código acima cria um objeto do tipo Carro e atribui sua referência à variável v. Note que essa variável v é uma variável que armazena referências para objetos do tipo Veículo. Esse tipo de atribuição é perfeitamente possível, já que um Carro é uma subclasse de Veículo. Através da variável v, é possível chamar os métodos que pertencem ao tipo Veículo.

Portanto, pode-se utilizar esse artifício de nomeação para transformar uma classe-filha em qualquer uma de suas classes-pai. *Professor, é possível fazer o inverso? Pode-se atribuir uma classe-pai a uma classe-filha?* Não, isso só pode ser feito por meio de um **type cast**. Assim, uma variável pode assumir momentaneamente outro tipo para que o programador possa utilizá-la.

3.7 JAVA: ENCAPSULAMENTO

Pessoal, nós já sabemos que para se descobrir o que um objeto pode fazer, basta olhar para as assinaturas de seus métodos públicos definidos na classe desse objeto – que formam uma interface de uso. A assinatura de um método é composta pelo seu nome e seus parâmetros. Por outro lado, para descobrir como um objeto realiza suas operações, deve-se observar o corpo de cada um dos métodos da classe.

Os corpos dos métodos constituem a implementação das operações dos objetos. *Professor, por que nós encapsulamentos classes, atributos e métodos?* Cara, por duas razões: desenvolvimento e manutenibilidade. O encapsulamento ajuda a aumentar a divisão de responsabilidades (ou coesão) e, dessa forma, fica mais fácil e rápido desenvolver sistemas em módulos.

Da mesma forma, ele ajuda a manutenção, visto que para torna-se mais difícil fazer "besteiras" no código e atrapalhar manutenções futuras. Trazendo isso para a vida real, lidamos com encapsulamento o tempo inteiro. Você sabe como usar um controle remoto, mas você não sabe como ele funciona internamente. Encapsula-se seu funcionamento interno e disponibiliza-se apenas sua interface ao usuário.

Chegamos ao conceito de Modificadores de Acesso! Eles são utilizados para modificar o modo como classes, métodos e variáveis são acessadas. Existem três modificadores de acesso e um quarto nível (acesso **default/friendly**), quando não se usa nenhum dos modificadores citados. Toda classe, método e variáveis de instância declaradas possuem um controle de acesso.

Pessoal, esses Modificadores de Acesso determinam quão acessíveis são esses elementos. Vamos vê-los agora em mais detalhes:

- **<public>**: essa instrução indica que a classe, método ou atributo assim declaradas podem ser acessadas em qualquer lugar e a qualquer momento da execução do programa – é o modificador menos restritivo.

- **<private>**: essa instrução indica que métodos ou atributos (classes, não) assim declaradas podem ser acessadas apenas dentro da classe que os criou. Subclasses herdam-nos, mas não os acessam – é o modificador mais restritivo[3].

[3] Em tese, no Mundo Java, classes não herdam nem acessam membros privados – objetos herdam membros privados, mas não o acessam; no Mundo OO, classes e objetos herdam membros privados, mas não o acessam. A documentação oficial afirma da seguinte forma: "*Members of a class that are declared private are not inherited by subclasses of that class*".

- **<protected>:** essa instrução indica que métodos ou atributos (classes, não) assim declaradas somente podem ser acessadas dentro do pacote em que está contida ou por subclasses no mesmo pacote.

- **<default>:** também chamado **friendly**, não há palavra para esse modificador porque ele é, na verdade, a ausência de um modificador. Indica-se que a classe, método ou atributo podem ser acessadas por classes do mesmo pacote.

Especificador	Própria Classe	Subclasse	Pacote	Global
Privado (-)	Sim	Não	Não	Não
<Vazio> (~)	Sim	Não*	Sim	Não
Protegido (#)	Sim	Sim	Sim	Não
Público (+)	Sim	Sim	Sim	Sim

OBSERVAÇÃO

É importante ressaltar que, em caso de não haver modificador, a subclasse pode ou não acessar os métodos e atributos da sua superclasse, e isso depende da localização da subclasse. Se ela estiver em um pacote diferente do pacote da superclasse, não poderá acessar. Se estiver em um mesmo pacote da superclasse, poderá acessar. Logo, para diferenciar o Modificador Pacote do Modificador Protegido, deve-se saber primeiramente se é desejável que a subclasse possa ter acesso a atributos e métodos da classe.

Pensem comigo! Acessar ou editar propriedades de objetos, manipulando-as diretamente, pode ser muito perigoso e gerar muitos problemas. Por conta disso, é mais seguro, para a integridade dos objetos e, consequentemente, para a integridade da aplicação, que esse acesso ou edição sejam realizados através de métodos desse objeto.

Utilizando métodos, podemos controlar como consultas e modificações são realizadas, controlando-as. Para tal, podemos utilizar Métodos Getters e Setters – para recuperar dados e inserir dados, respectivamente. Para o primeiro, utiliza-se o Método **Get**; para o segundo, utiliza-se o Método **Set**. Em geral, costuma-se declarar atributos como privados, e métodos e classes como públicos.

```
class Classe1 {

  //Atributo privado
  private String Algo;
```

```
    //Método público para recuperar dados
    public String getAlgo() {
        return this.Algo;
    }
    //Método público para modificar/inserir dados
    public void setAlgo(String Algo) {
        this.Algo = Algo;
    }
}
```

3.8 JAVA: INTERFACE

Galera, o que é uma Interface? É simplesmente um contrato! Quando vocês assinam o contrato do seguro de um carro, vocês estão se comprometendo a atender aquilo que lá está escrito. Analogamente, a interface é um contrato que obriga aqueles que a assinam a implementar os métodos lá presentes. Elas ajudam a padronizar implementações – entradas e saídas.

Em outras palavras, é um recurso utilizado em Java para obrigar a um determinado grupo de classes a ter métodos ou propriedades em comum para existir em um determinado contexto, contudo os métodos podem ser implementados em cada classe de uma maneira diferente. Em geral, as interfaces são compostas basicamente de um conjunto de assinaturas de métodos públicos e abstratos.

```
public interface FiguraGeometrica
{
   public String getNomeFigura();
   public int getArea(int vertice);
   public int getPerimetro();
}
```

A sintaxe para implementar uma Interface utiliza a palavra reservada implements:

```
public class ClasseImplementadoraDeInterfaces implements FiguraGeometrica
```

Professor, qual a diferença entre uma Interface e uma Classe Abstrata? Bem, Interfaces não são classes; são, na verdade, entidades que não possuem qualquer implementação, apenas assinatura, sendo que todos os seus métodos são públicos e abstratos. Já as Classes Abstratas também contêm, em geral, métodos abstratos (sem corpo), mas podem ter vários métodos concretos.

Uma Classe Abstrata pode, inclusive, não conter nenhum método abstrato, i.e., todos os seus métodos são concretos. No entanto, se uma classe tiver um único método abstrato que seja, ela será considerada uma Classe Abstrata. Aliás, uma Interface é também chamada de classe abstrata pura por conta disso, ou seja, não há impurezas (isto é, métodos concretos).

3.9 JAVA: POLIMORFISMO

A palavra Polimorfismo vem do grego: muitas formas. Trata-se da capacidade de um objeto poder se comportar de diversas formas dependendo da mensagem recebida. Observem que isso não quer dizer que o objeto fica transformando seu tipo a todo momento. Na verdade, um objeto nasce com um tipo e morre com esse mesmo tipo. *O que muda, então?* É a forma como nós nos referimos a esse objeto!

Existem dois tipos de polimorfismo:

- Polimorfismo Estático: ocorre quando uma classe possui métodos com mesmo nome, entretanto assinaturas diferentes, i.e., métodos de uma mesma classe se sobrecarregando. Pode ser chamada também de Sobrecarga ou *Overloading*. Ocorre em Tempo de Compilação e alguns não o consideram um tipo de polimorfismo, porque a assinatura é diferente.

- Polimorfismo Dinâmico: ocorre quando uma classe possui um método com mesmo nome e mesma assinatura que um método de sua suplerclasse, i.e., o método da classe-filha sobrescreve o método da classe-pai. Pode ser chamada também de Sobrescrita, *Overriding*, Redefinição ou Sobreposição. Ocorre em Tempo de Execução e é um corolário do conceito de herança.

Professor, o que você quer dizer com mesma assinatura e assinatura diferente? É a mesma quantidade, tipo e ordem dos parâmetros. Em outras palavras:

```
//Assinatura Igual: quantidade, tipo e ordem
public String EntendendoAssinatura(int A, char B);
public String EntendendoAssinatura(int C, char D);

//Assinatura Diferente: quantidade diferente
public String EntendendoAssinatura(long A, long B, long C);
public String EntendendoAssinatura(long A, long B);

//Assinatura Diferente: tipo diferente
public String EntendendoAssinatura(long A, long B);
public String EntendendoAssinatura(char A, long B);

//Assinatura Diferente: ordem diferente
public String EntendendoAssinatura(int A, char B);
public String EntendendoAssinatura(char B, int A);
```

Agora vamos ver um exemplo de Polimorfismo Dinâmico. Eu pensei comigo mesmo: *O que seria uma característica comum de praticamente todos os animais?* Emitir sons! Observem que eu criei uma classe abstrata que possui um único método – também

abstrato –, que não retorna valor algum e não recebe nenhum argumento. Vejam a classe abaixo:

```
abstract class Animal {
   abstract void som();
}
```

Dito isso, vou criar dois animais do meu gosto pessoal: um gato e um cachorro! Bem, o gato é um animal! *Que relacionamento é esse "é um"?* Herança! Portanto as classes gato e cachorro serão classes filhas da superclasse Animal. Observem abaixo que ambas implementam o método **abstract void som( )**, porém cada uma a sua maneira, visto que gatos e cachorros emitem sons diferentes!

```
class Gato extends Animal {
   void som() {
      System.out.println("MIAU!");
   }
}

class Cachorro extends Animal {
   void som() {
      System.out.println("AUAU!");
   }
}
```

Pois bem! Vamos ver agora o Polimorfismo Dinâmico em ação. Criaremos um objeto do tipo Gato e atribuiremos a um objeto do tipo Animal. *Professor, você pode fazer isso?* Sim, porque Gato é filho de Animal – é similar a um *casting* implícito! Em seguida chamaremos o método **som( )**. Por fim, faremos o mesmo procedimento com o objeto do tipo Cachorro.

```
public static void main(String[] args) {
   Animal a1 = new Gato();
   a1.som(); //Emite o som MIAU!
   Animal a2 = new Cachorro();
   a2.som(); //Emite o som AUAU!
}
```

Olha que bacana: existem dois métodos com exatamente o mesmo nome e mesma assinatura! *Como o compilador saberá qual deve ser chamado?* Ele não saberá – tem que ser em tempo de execução. No primeiro momento, ele apresentará "MIAU!", porque animal é nesse instante um gato. Depois fazemos outra atribuição e ele apresentará "AUAU", porque animal naquele instante é um cachorro.

O Polimorfismo Estático é bem mais simples! Imaginem que eu deseje fazer dois cálculos matemáticos. Primeiro somar três números e depois somar apenas dois números. Eu posso ter dois métodos com mesmo nome, mas assinaturas diferentes.

Dessa forma, se eu passar três valores, ele saberá que é um método; e se eu passar dois valores, ele saberá que é outro método. *Simples, não?*

```
class Calculo {

   void soma(int a,int b){
      System.out.println(a+b);
   }
   void soma(int a,int b,int c){
      System.out.println(a+b+c);
   }

   public static void main(String args[]) {
      Calculo x = new Calculo();
      x.soma(10,10,10);  //Mesmo nome, mas assinatura diferente (3 valores)
      x.soma(20,20);     //Mesmo nome, mas assinatura diferente (2 valores)
    }
}
```

OBSERVAÇÃO

Atributos com o mesmo nome na classe/subclasse substituem os herdados. Ademais, métodos declarados com a palavra-reservada final não podem ser redefinidos. Já os métodos abstratos devem ser redefinidos na subclasse ou declarados como abstratos para que sejam implementados pela classe-neta. Por fim, membros definidos na superclasse podem ser acessados na subclasse por meio da palavra-reservada super, a menos que tenham sido declarados como privados.

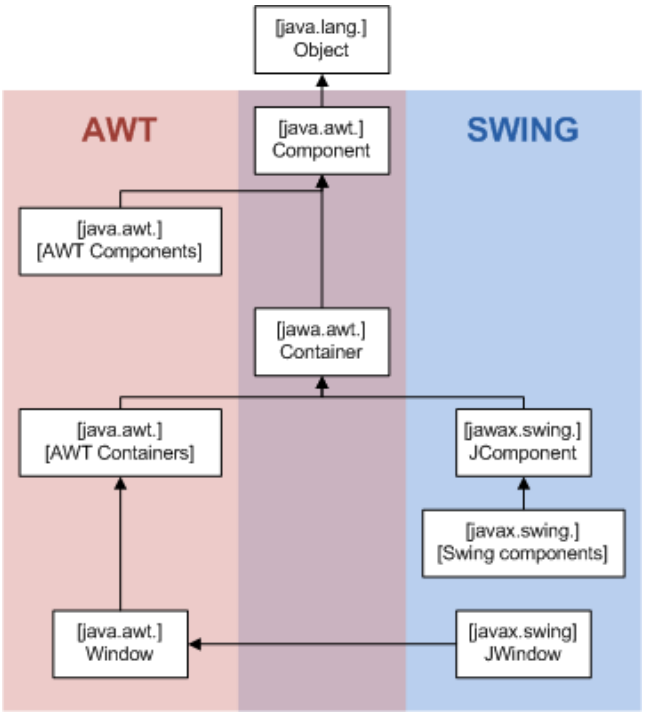
4 JAVA: CONCEITOS AVANÇADOS

4.1 JAVA: INTERFACE GRÁFICA

A Plataforma Java oferece recursos para construção de interfaces gráficas de usuário (GUI), entre eles: AWT (**java.awt**) e Swing (**javax.swing**)! O primeiro é um conjunto básico de classes e interfaces que definem os componentes de uma janela desktop. Já o Swing é um conjunto sofisticado de classes e interfaces que definem os componentes visuais necessários para construir uma interface gráfica de usuário.

Vocês entenderam mais ou menos? O primeiro é um conjunto básico que serve de base para o segundo, que é um conjunto mais sofisticado. Os componentes Swing são implementados com nenhum código nativo – totalmente Java puro, i.e., apesar de serem sensivelmente mais lentos que os componentes nativos em AWT (Abstract Window Toolkit), eles oferecem uma maior liberdade aos programadores.

O AWT (**javax.awt**) veio primeiro, é mais pesado, é gerado pelo sistema operacional, logo é dependente de plataforma. O Swing (**javax.swing**) é mais leve, é gerado por uma Máquina Virtual Java (JVM), logo é independente de plataforma. Galera, nem tudo é diferente! Vejam: ambos são fáceis de programar, porque a orientação a objetos proporciona alterar partes do programa, sem alterar toda a estrutura.



O Swing mantém as funcionalidades dos componentes AWT! Ao acrescentarmos a letra "J" aos componentes AWT, as novas classes serão como componentes Swing,

i.e., JButton, JTextField, JList tem os mesmos argumentos que Button, TextField e List – componentes AWT. Nós vamos ver logo à frente a definição de alguns componentes Swing. *Professor, o que é um componente?*

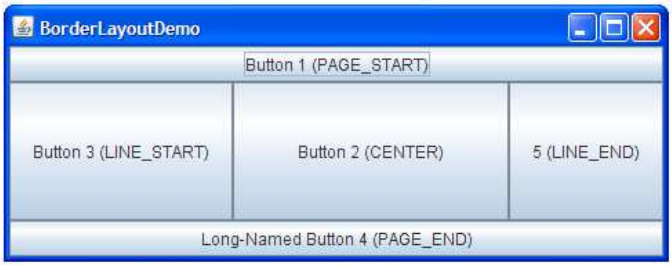
Bem, os itens que aparecem em uma interface gráfica de interação com usuário (janelas, caixas de texto, botões, listas, caixas de seleção, entre outros) são chamados de componentes. Alguns componentes podem ser colocados dentro de outros componentes, por exemplo, uma caixa de texto dentro de uma janela. Abaixo podemos ver a definição básica dos principais componentes:

- JFrame: define janelas com título, borda e alguns itens definidos pelo sistema operacional como botão para minimizar ou maximizar;
- JPanel: define um componente que basicamente é utilizado para agrupar nas janelas outros componentes como caixas de texto, botões, listas, entre outros;
- JTextField: define os campos de texto menores que podem ser preenchidos pelo usuário;
- JTextArea: define os campos de texto maiores que podem ser preenchidos pelo usuário;
- JPasswordField: define os campos de caixa de texto de formulários para digitar senhas;
- JButton: permite que os usuários indiquem quais ações ele deseja que a aplicação execute;
- JCheckBox: permite criar formulários com checkbox's (aquelas caixinhas para você dar um check);
- JComboBox: permite criar formulários com combobox's (aquela listinha de opções).

Agora vamos falar um pouquinho sobre o Layout Manager (ou Gerenciador de Disposição)! Galera, uma coisa é criar diversos componentes, outra coisa é posicioná-los e dimensioná-los. O Layout Manager é o objeto que determina como elementos e componentes serão dispostos em tela, tamanhos, comportamentos, entre outros aspectos.

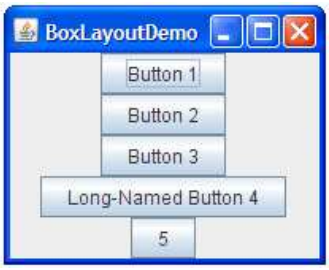
Ele controla os componentes que estão dentro do componente ao qual ele está associado. Os principais Layout Managers são:

BorderLayout	Divide a área de um componente de background em cinco regiões (norte, sul, leste, oeste e centro). Somente um componente pode ser adicionado em cada região. Eventualmente, o BorderLayout altera o tamanho preferencial dos componentes para torná-los compatíveis com o tamanho das regiões. O BorderLayout é o Layout Manager padrão de um JFrame.

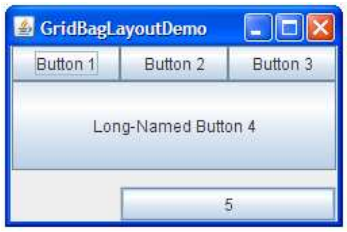


FlowLayout	Arranja os componentes da esquerda para direita e quando o tamanho horizontal não é suficiente ele "pula" para a próxima "linha". O FlowLayout não altera o tamanho preferencial dos componentes. O FlowLayout é o Layout Manager padrão de um JPanel (Swing).

BoxLayout	Arranja os componentes de cima para baixo "quebrando linha" a cada componente adicionado. O BoxLayout não altera o tamanho preferencial dos componentes.



GridLayout	Divide a área de um componente de background em células semelhantemente a uma tabela. As células possuem o mesmo tamanho.



GridBagLayout	É o mais complexo layout e é baseado no GridLayout. A ideia é representar a tela como um grid com linhas e colunas, mas podemos posicionar elementos ocupando várias células em qualquer direção, o que permite layouts mais customizados (apesar do alto custo de manutenção). A definição de onde deve ser colocado cada componente é feita através de restrições (GridBagConstraints) passadas ao método add.

Professor, e o Look and Feel (L&F)? O "Look" se refere a aparência e o "Feel" se refere ao comportamento dos componentes. É como se fosse um skin, um tema, customizável ou não. O Java oferece algumas opções: **CrossPlatformLookAndFeel** (ou **Metal**), que parece o mesmo em qualquer plataforma; **SystemLookAndFeel**, que usa o L&F nativo do sistema utilizado; e **Synth**, que permite criar novos.

Pessoal, a principal função de uma interface gráfica de usuário é permitir interação entre usuários e aplicação. Os usuários interagem com uma aplicação clicando em botões, preenchendo caixas de texto, movimentando o mouse, entre outros. Essas ações dos usuários disparam eventos que são processados pela aplicação através de *Event Listeners*.

Você pode descobrir quais tipos de eventos um componente pode disparar ao olhar seus tipos de Event Listeners! Para criar um *Listener*, devemos implementar a interface correspondente ao tipo de evento que queremos tratar. Por exemplo:

- KeyListener: utilizado quando se deseja tratar eventos de pressionar ou soltar teclas do teclado.

- MouseListener: utilizado quando se deseja tratar eventos como cliques dos botões do mouse (ex: duplo-clique, clique-arrasta, etc).

- WindowsEvent: utilizado quando se deseja tratar eventos que envolvem a manipulação de janelas.

As tarefas de respostas realizadas em um evento são conhecidas como *Handler* de evento e o processo total de responder a eventos é conhecido como tratamento de evento. Para cada tipo de evento precisa ser implementada uma interface de escuta. Quando um evento acontece, é realizado o despacho (dispatching) para os ouvintes apropriados.

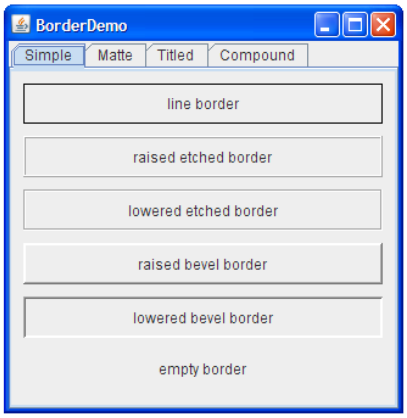
Esse despacho chama um método de tratamento de evento em cada um de seus ouvintes, sendo registrados para o tipo de evento ocorrido. A ocorrência de um evento faz com que o componente receba um ID único de evento – ele especifica o tipo de evento. Então, o componente pega esse ID para decidir qual tipo de ouvinte será útil, decidindo qual o método que vai chamar para cada objeto listener.

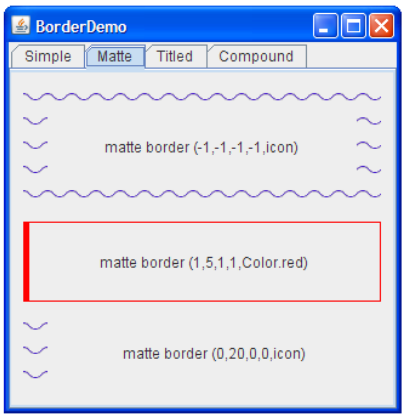
O evento é despachado por todas as decisões que são tratadas para o usuário através dos componentes GUI. Sendo necessário ser feito, pois precisa ser registrado um handler de evento para o tipo particular de evento que o aplicativo exige. O componente vai assegurar que o método apropriado do handler de evento é chamado quando o evento ocorrer.

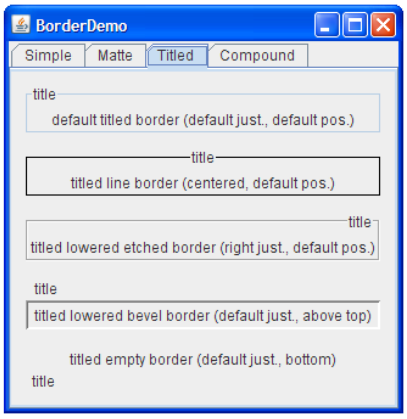
Pessoal, todo JComponent pode ter um ou mais bordas. As bordas são objetos inacreditavelmente úteis que, não sendo componentes, sabem como desenhar as

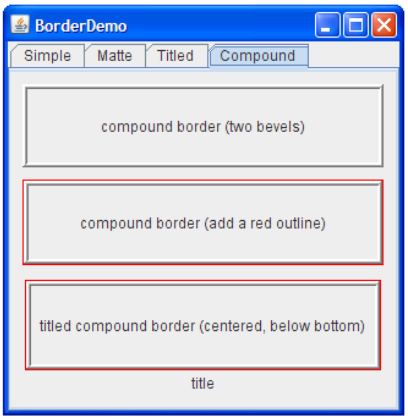
margens dos componentes. Elas são úteis não só para desenhar linhas e margens elegantes, mas também para fornecer títulos e espaços vazios em componentes. Para colocar uma borda em um JComponent, deve-se usar o método **setBorder**.

Pode-se usar a classe **BorderFactory** para criar a maioria das bordas fornecidas. Se você precisar de uma referência para determinada uma borda, você pode salvá-la em uma variável do tipo **Border**, que conterá sua borda customizada. Abaixo podemos ver uma borda linear e, em seguida, vários outros tipos de bordas diferentes! Observem a diversidade disponível...









4.2 JAVA: TIPOS ENUMERADOS

Trata-se de um tipo de dados especial que habilita uma variável a ser um conjunto pré-definido de constantes. A variável deve ser igual a um dos valores que foram predefinidos para ela. Exemplos comuns incluem direções de uma bússola ou os dias da semana. *Por que?* Porque são valores constantes! Ademais, lembrem-se que eles sempre vêm em letra maiúscula. A sintaxe básica inclui a palavra-reservada `enum`:

```
public enum nomeENUM {
  <lista de constantes>
}
```

Para especificar os dias da semana, as direções de uma bússola ou os planetas do sistema solar em um tipo `enum`, podemos fazer:

```
public enum Dia {
  SEGUNDA, TERÇA, QUARTA, QUINTA,
  SEXTA, SÁBADO, DOMINGO; }

public enum Bussola {
  NORTE, SUL, LESTE, OESTE; }

public enum Planetas {
  MERCÚRIO, VENUS, TERRA,
  MARTE, JUPTER, SATURNO,
  URANO, NETUNO, PLUTÃO; }
```

Professor, qual é? O que tem demais em uma lista de constantes? Calma, amigão! Nós podemos adicionar alguns valores a essas constantes, mas para isso devemos primeiro declarar um construtor para, então, inicializar os atributos com os valores. Por exemplo, a distância para o sol em milhões de quilômetros! O construtor tem apenas um argumento porque a constante tem apenas um valor. *Bacana?*

```
public class Main {

  public enum Planetas {
    MERCURIO(57), VENUS(108), TERRA(149),
    MARTE(227), JUPITER(778), SATURNO(1429),
    URANO(2870), NETUNO(4504), PLUTAO(5913);

    public int distanciaSol;

    Planetas(int distanciaSol) {
      this.distanciaSol = distanciaSol; }
  }
    public static void main(String[] args) {
    System.out.println("O Planeta mais afastado é: " + Planetas.PLUTAO); }
  }
```

4.3 JAVA: ANOTAÇÕES

O Java 5 nos trouxe uma grande novidade que praticamente revolucionou o desenvolvimento de software! As Anotações permitem declarar metadados dos objetos nos próprios objetos e, não, em um arquivo separado. Dessa forma, configurações de uma classe poderiam permanecer dentro da própria classe, em vez de ficarem, por exemplo, em um Arquivo de Configuração XML.

Professor, por que arquivos de configuração são tão detestáveis? Cara, porque muitas vezes eles tornam extremamente difíceis a compreensão de alguns sistemas. As anotações são mais simples, discretas, compreensíveis e podem efetivamente ajudar na automatização de algumas tarefas. Arquivos de Configuração são, algumas vezes, complexos, grandes, chatos e difíceis de entender.

A Anotação, como o próprio nome diz, é uma forma de anotar, marcar, apontar classes, campos ou métodos, de tal maneira que essas marcações possam ser tratadas por um compilador, ferramentas de desenvolvimento e bibliotecas. Ela provê dados sobre um programa, mas não faz parte dele em si, isto é, elas não afetam diretamente a operação do código que elas anotam.

Elas podem fornecer informações sobre o código que está sendo escrito ou até mesmo do próprio programa, semelhante a comentários. No entanto, elas podem ser utilizadas como um objeto semântico de compiladores, isto é, facilita bastante a vida dos compiladores. Dessa forma, eles podem entender que, por exemplo, não é para mostrar mensagens de advertências (os famosos *warnings*).

Eles podem, inclusive, utilizar anotações para detectar erros de código; ou mesmo para criar documentações por meio de XML. Algumas anotações podem ser avaliadas em tempo de execução e podem possuir elementos ou não. *Professor, qual é a sintaxe básica de uma anotação?* As anotações são sempre precedidas de arroba (@). É absurdamente simples:

```
@annotation
```

Por convenção, elas vêm antes do elemento que se deseja anotar (Ex: Antes do método ou classe). Abaixo temos uma lista com as anotações mais utilizadas:

- @Deprecated: indica que um método tem seu uso desencorajado por ser perigoso ou por ter uma alternativa melhor desenvolvida;

- @Override: indica que um método da classe-pai será sobrescrito por um método da classe-filha;

- @SupressWarning("unchecked"): indica que todos os avisos ou *warnings* da categoria "não verificado" devem ser ignorados;

- Existem mais de sessenta atualmente: @SafeVarargs, @FunctionalInterface, @Retention, @Documented, @Target, @Inherited, @Repeatable, @id.

Anotações podem ser de três tipos:

- Anotações Marcadoras: são aquelas que não possuem membros; são identificadas apenas pelo nome, sem dados adicionais (Por exemplo: @id – não há valores ou dados adicionais).

- Anotações de Valor Único: são aquelas que possuem um único membro, o próprio valor; dessa forma, não é necessário informar o nome (Por exemplo: @SupressWarning("unchecked") é equivalente a @SupressWarning(value = "unchecked")).

- Anotações Completas: são aquelas que possuem múltiplos membros; assim, deve-se usar a sintaxe completa para cada par nome/valor (Por exemplo: @Version(major = 1, minor = 0, micro = 0)).

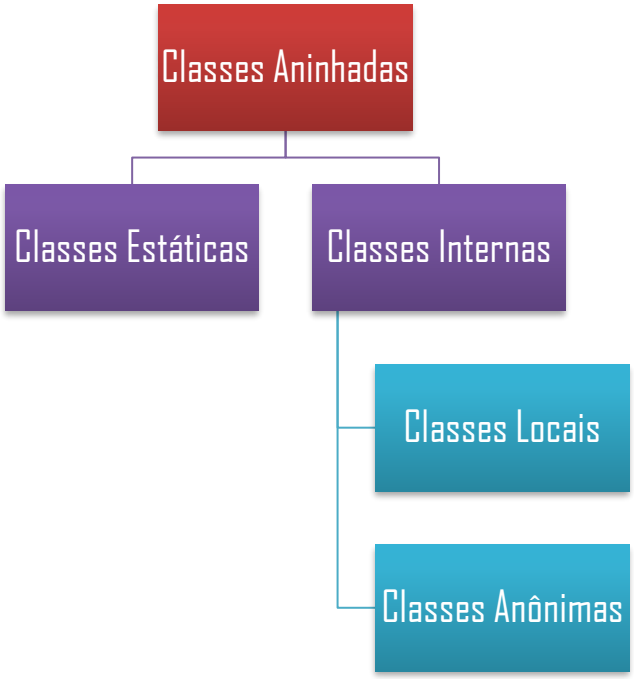
E que tal inventar sua própria anotação? Cara, é muito fácil e semelhante a interfaces! Tão parecido que se utiliza a mesma palavra, mas precedida de um @. Vejamos:

```
<modificador> @interface identificador {
   <Declaração de Elementos>
}
```

Os modificadores e os elementos são opcionais! O identificador não pode ser o mesmo de uma classe ou interface utilizadas na aplicação. No código abaixo, criamos uma anotação que possui apenas um valor. Para chamá-la, basta utilizar o comando @MinhaAnnotation. É possível, inclusive, adicionar sua anotação personalizada no Javadoc. *Bacana, né?!*

```
public @interface MinhaAnnotation {
   int valor() default 10;
}
```

4.4 JAVA: CLASSES INTERNAS (ANINHADAS)



Java permite definir uma classe dentro de outra classe. Essa classe é conhecida como Classe Aninhada e é ilustrada acima! Elas se dividem em duas categorias: estáticas e não-estáticas. Classes Aninhadas que são declaradas com **static** são chamadas Classes Aninhadas Estáticas. Já as Classes Aninhadas Não-Estáticas são chamadas mais comumente de Classes Internas – como podemos ver abaixo:

```
class ClasseExterna {
   ...
   static class ClasseAninhadaEstatica {
       ...
   }
   class ClasseInterna {
       ...
   }
}
```

Uma Classe Aninhada é um membro da Classe Externa. Classes Aninhadas Não-Estáticas (ou Classes Internas) possuem acesso aos membros da Classe Externa, mesmo que eles sejam declarados como **private**. Classes Aninhadas Estáticas não possuem acesso aos membros da Classe Externa e podem ser declaradas como **private**, **public**, **protected** ou **default** (sem modificador).

As Classes Externas só podem ser declaradas como **public** ou **default**. *Aí vocês devem estar se perguntando: por que usar classes aninhadas?* Cara, é uma maneira de agrupar logicamente classes que são utilizadas em apenas um lugar. Se você possui

uma Classe B que com certeza será usada apenas dentro da Classe A, o melhor é criar a Classe B como interna a Classe A.

Além disso, ela aumenta o encapsulamento. Imaginem que A e B são classes normais (não-aninhadas, também chamadas top-level) e a Classe B precisa acessar os membros da Classe A. Podemos, então, colocar a Classe B dentro da Classe A e colocar os membros de A como privados. Ainda assim, a Classe B poderia acessar os membros da Classe A por ser uma classe aninhada.

Por fim, ela pode levar a códigos mais legíveis e fáceis de dar manutenção! Colocar classes pequenas dentro de classes não-aninhadas, já que apenas essa a utilizará, faz com que a lógica da classe não-aninhada seja mais fácil de ser identificada, consequentemente tornando o código mais legível e de fácil manutenção. Essas são apenas algumas das vantagens de se utilizar classes aninhadas.

Bem como métodos e variáveis de classe, uma classe aninhada estática é associada à sua classe exterior. E assim como métodos de classes estáticas, uma classe aninhada estática não pode se referir diretamente a variáveis ou métodos de instância definidos na Classe Exterior. Elas são acessadas utilizando o nome da classe externa: ClasseExterna.ClasseAninhadaEstatica.

Vamos falar um pouco sobre as Classes Internas! Da mesma forma que métodos e variáveis de instância, uma classe interna é associada a uma instância de sua classe externa e tem acesso direto a métodos e campos desse objeto. Além disso, como uma classe interna é associada a uma instância, ela não pode definir nenhum membro estático.

Objetos que são instâncias de uma classe interna existem dentro da instância de uma classe externa. *Entenderam isso?* Uma instância da classe interna só pode existir dentro de uma instância da classe externa, e tem acesso direto aos métodos e campos de sua instância externa. Para instanciar uma classe interna, deve-se primeiro instanciar a classe externa, como segue:

```
ClasseExterna.ClasseInterna objetoInterno = objetoExterno.new ClasseInterna();
```

As classes internas se dividem em classes locais e anônimas! As primeiras são classes definidas em um bloco (grupo de zero ou mais declarações entre chaves). Em geral, classes locais são encontradas no corpo de um método. As segundas permitem escrever códigos mais concisos, e declarar e instanciar uma classe ao mesmo tempo. São comuns quando se deseja usar uma classe local apenas uma vez.

4.5 JAVA: REFLEXÃO E GENÉRICOS

Reflexão (ou Reflection) é comumente utilizada por programas que requerem a habilidade de examinar ou modificar o comportamento em tempo de execução de aplicações que rodam em uma Java Virtual Machine (JVM). Em geral, é um recurso bastante avançado e extremamente poderoso, e deve ser utilizado apenas por programadores experientes.

Reflection pode permitir que aplicações executem operações que, por muito tempo, se pensou impossível. Ele permite criar chamadas em tempo de execução, sem precisar conhecer as classes e objetos envolvidos quando escrevemos nosso código. Esse dinamismo é necessário para resolvermos tarefas que nosso programa só descobre serem necessárias ao receber dados, em tempo de execução.

Essa tecnologia possibilita listar todos os atributos de uma classe e pegar seus valores em um objeto; instanciar classes cujo nome só vamos conhecer em tempo de execução; invocar métodos dinamicamente baseado no nome do método como String; descobrir se determinados pedaços do código têm annotations. É um recurso muito poderoso!

Já o Generics permite que você personalize um método ou uma classe genérica para qualquer tipo que você esteja trabalhando. Para ter certeza da tipagem dos objetos em tempo de compilação, devemos aplicar o recurso do Generics. Com este recurso podemos determinar o tipo de objeto que queremos armazenar em uma coleção no momento em que ela é criada.

A partir daí, o compilador não permitirá que elementos não compatíveis com o tipo escolhido sejam adicionados na coleção. Isso garante o tipo do elemento no momento em que ele é recuperado da coleção e elimina a necessidade de casting. Imaginem que bacana seria poder escrever um único método de ordenação de elementos de um vetor.

Mas não só isso, esse método seria capaz de ordenar tanto um vetor de inteiros como um vetor de String, ou qualquer outro tipo. Nossa tecnologia permite que programadores especifiquem, com uma simples declaração de método, um conjunto de métodos relacionados; ou, com uma simples declaração de classes, um conjunto relacionado de tipos respectivamente.

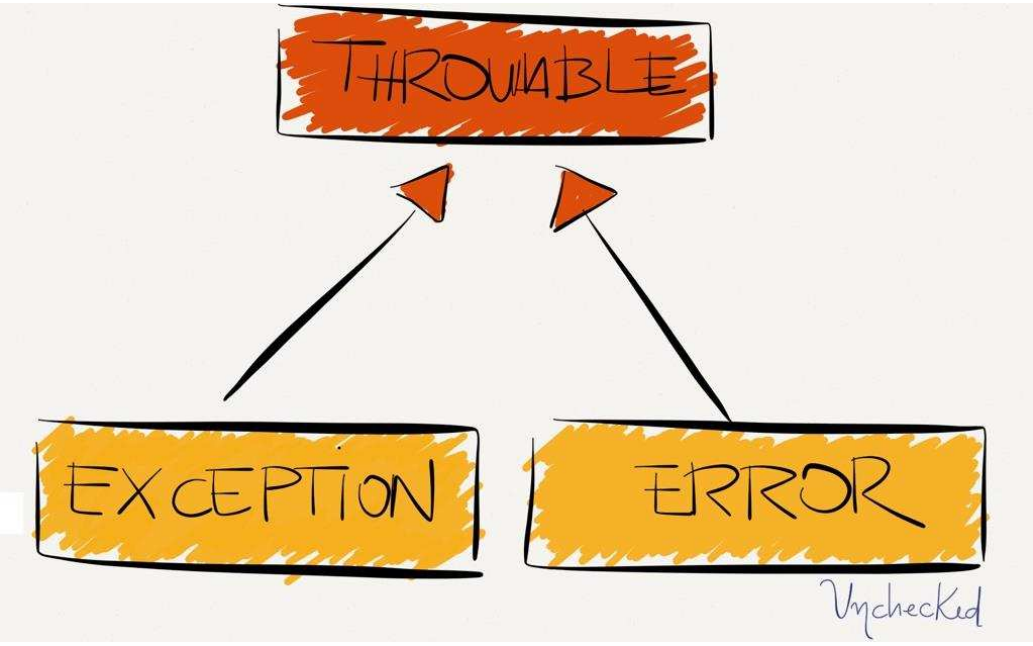
4.6 JAVA: TRATAMENTO DE EXCEÇÕES

Em programação, há sempre a possibilidade de ocorrer erros imprevistos durante a execução de um sofware, esses erros são exceções e podem ser provenientes de erros de lógica, acesso a dispositivos ou arquivos inexistentes, edição de algo sem permissão, etc. Na Linguagem C, códigos de erro são utilizados para indicar o tipo de erro que ocorreu! *Como assim, professor?* Vejamos o exemplo abaixo:

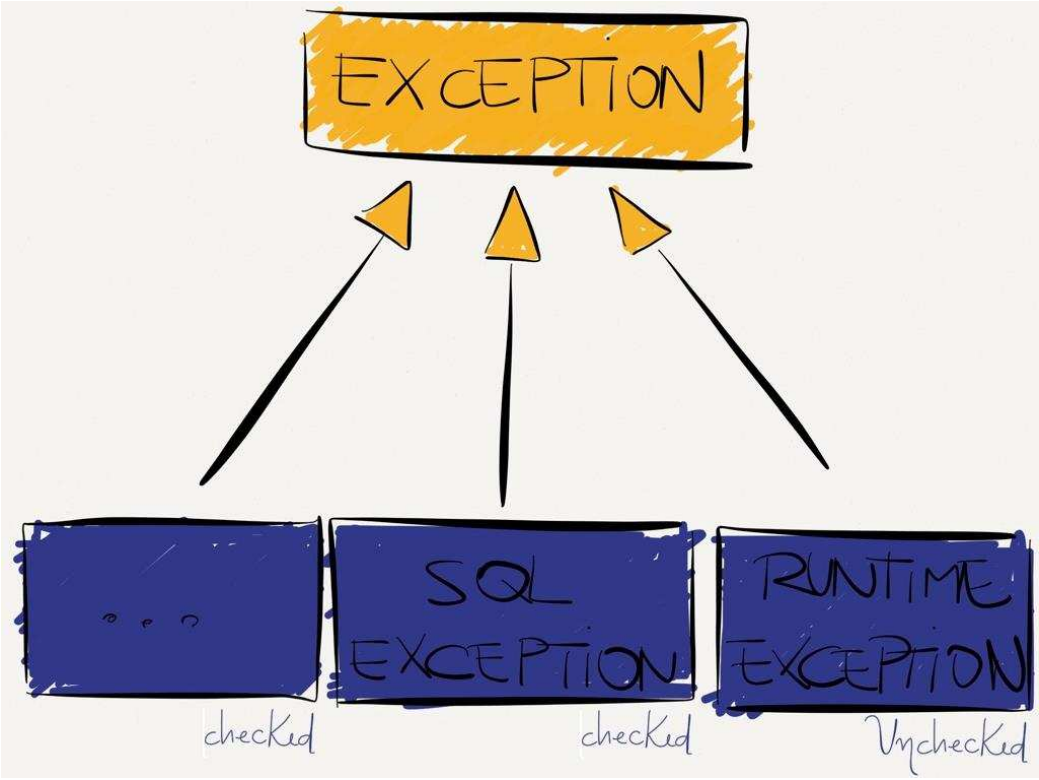
```
int testeErro(int idade) {
   if(idade >= 0)
      idade = idade + 18;
   else
      return 100; //Código de Erro para valor negativo
}
```

Observem que se a idade informada for maior ou igual a zero, realiza-se uma determinada operação; se for menor que zero (negativa), retorna-se o Código de Erro 100 – para que o programador saiba o que ocorreu. *Qual o problema dessa abordagem?* Primeiro, exige uma vasta documentação indicando o que significa cada código de erro (Ex: 100 = Valores negativos; 200 = Sem permissão de escrita).

Além disso, observe que a palavra-reservada **return** fica ocupada, impossibilitando a devolução de outros possíveis resultados. *Entenderam essa parte?* Eu poderia usar essa palavra para retornar valores importantes para o código, mas não posso porque ele está ocupado em informar se houve ou não erro. Pois é, o Java possui uma estratégia diferente para contornar esses imprevistos.



Ele busca realizar o tratamento dos locais do código que podem vir a lançar possíveis exceções. Java possui a classe **Throwable**, que modela todos os tipos de erros de execução e que se divide em duas subclasses **Error** e **Exception**. A primeira define erros que não devem ser capturados pelas aplicações, pois representam erros graves que não permitem que a execução continue de maneira satisfatória[4].



A segunda define erros para os quais as aplicações normalmente têm condições de realizar um tratamento, logo **Exception** e **Error** são subtipos de **Throwable**. As exceções ainda se dividem em verificadas (**Checked**), quando obrigatoriamente devem ser tratadas e não-verificadas (**Unchecked**), quando não há essa obrigação – programador decide! *E como se detectam, manipulam e tratam as exceções?*

Bem, sempre que um método de alguma classe for passível de causar algum erro previsto, nós podemos utilizar um método de tentativa chamado **try**. Tudo que estiver dentro do bloco **try** será executado até que alguma exceção seja lançada, ou seja, até que algo dê errado. Quando uma exceção é lançada, ela sempre deve ser capturada. O trabalho de captura da exceção é executado pelo bloco **catch**.

Podemos encadear vários blocos **catch**, dependendo do número de exceções que podem ser lançadas por uma classe ou método. O bloco catch obtém o erro criando uma instância da exceção. Quando uma exceção é lançada e é necessário que

[4] Exemplo: estouro de memória.

determinada ação seja tomada mesmo após a sua captura, utilizamos a palavra reservada finally – é opcional, mas se existir, sempre será executado.

É útil para liberar recursos do sistema quando utilizamos, por exemplo, conexões de banco de dados e abertura de buffer para leitura ou escrita de arquivos. finally virá após os blocos de catch. Portanto, o **try** indica que um bloco de código pode ocorrer erro; o **catch** tem o objetivo de capturar, manipula e trata erros; e o **finally** busca realizar ações mesmo após a captura de erros. Vejamos a estrutura básica:

```
try { //Não vem sozinho: try/catch, try/finally ou try/catch/finally

  //Código a ser executado
} catch (ClasseDeExceção objDaExceção) { //Não vem sozinho: try/catch ou try/catch/finally

  //Tratamento da exceção
} finally { //Não vem sozinho: try/finally ou try/catch/finally.

  //Código a ser executado mesmo que uma exceção seja lançada
}
```

Algumas observações importantes: o **catch** deve aparecer após o **try** e entre os blocos não deve haver nenhuma outra instrução. Além disso, quando uma exceção é identificada no try, o restante do código não é executado e não há um retorno para o término do código. Implicitamente, todas as classes Java automaticamente lançam uma exceção de RuntimeException.

Por fim, vamos falar da propagação de exceções! Imagine uma situação em que não é desejado que uma exceção seja tratada na própria classe ou método, mas sim em outra classe ou método que venha lhe chamar. Para solucionar tal situação utilizamos o a cláusula **throws** na assinatura do método indicando explicitamente a possível exceção que o mesmo poderá a vir lançar. Sintaxe abaixo:

```
tipoDeRetorno nomeDoMetodo(...) throws tipoExcecao1, tipoExcecao2, tipoExcecaoN
```

Em outras palavras, utilizamos **throws** para indicar que qualquer um que chame aquele método deve tratar suas possíveis exceções. No entanto, algumas vezes podemos fazer com que um método lance um throwable (em geral, do tipo exceção) sem exigir que aqueles que chamem esse método tratem essa exceção – para tal, utilizamos throw. Sintaxe abaixo:

```
throw new Exception("Número não pode ser negativo!");
```

Observem que a palavra-reservada throw é utilizada dentro do corpo do método para invocar uma exceção qualquer personalizada explicitamente, além de não

poder lançar mais de uma exceção; já a cláusula **throws** é utilizada na declaração ou assinatura de um método e declaração uma exceção a ser tratada, além de poder lançar mais de uma exceção.

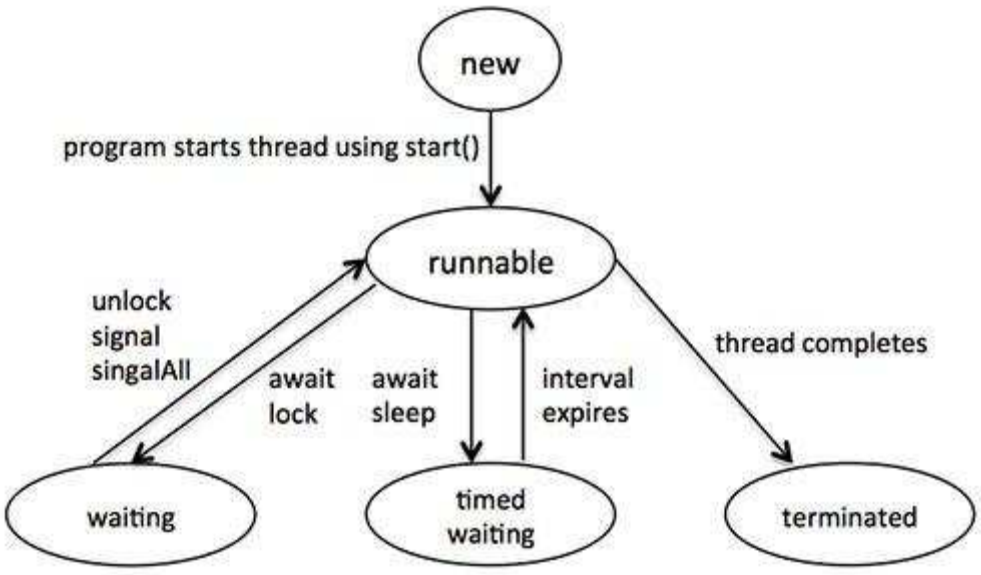
4.7 JAVA: SINCRONISMO E MULTITHREADING

Vocês já devem ter percebido que os programas que utilizamos corriqueiramente conseguem executar diversas atividades relativamente independentes entre si. Você está lá navegando em dezessete abas diferentes pelo Firefox e ao mesmo tempo está baixando algum arquivo, ouvindo alguma música, lendo a minha aula e enviando um e-mail – tudo paralelamente.

E aqueles sistemas que você utiliza no seu trabalho em que várias pessoas acessam para fazer coisas diferentes? Pois é, já que essas atividades são relativamente independentes entre si, elas podem ser executadas em paralelo. *Vocês concordam comigo?* Na verdade, a maioria dos softwares realizam diversas tarefas paralelamente sem nós percebermos!

Quando nós executamos essas tarefas em paralelo, estamos usando Threads! *Vocês sabem o que essa palavra significa em português?* Fios ou Linhas! Em outras palavras, criamos linhas de execução de tarefas paralelas em memória – cada linha responsável por executar alguma coisa simultaneamente e relativamente independentes. Em Java, as Threads são objetos presentes no Pacote **java.lang**.

Além da vantagem evidente de poder realizar tarefas simultaneamente, podemos dizer que assim utilizamos os recursos disponíveis de melhor forma, especialmente quando o computador em questão possui múltiplos processadores. Para entender isso melhor, podemos observar o ciclo de vida de threads e cada estágio pelo qual ela passa – do início ao fim!



- New: uma nova thread começa seu ciclo de vida no estado new e permanece nele até o programa inicializar a thread – é como se ela tivesse nascido.

- Runnable: após uma nova thread ter sido inicializada, a thread se torna runnable – é nesse estado que ela executa uma tarefa.

- Waiting: algumas vezes, uma thread espera outra thread realizar alguma tarefa até que esse thread sinalize que ela pode continuar sua execução.

- Timed Waiting: é o mesmo caso que o anterior, no entanto aqui o intervalo de tempo de espera é especificado.

- Terminated: uma thread runnable entra nesse estado quando completa sua tarefa ou quando termina.

Dissemos várias vezes que, em geral, threads são relativamente independentes. Nós dissemos dessa maneira, porque quando elas utilizam recursos em comum, é preciso haver um sincronismo. Quando muitas threads são executadas, é necessário sincronizar suas atividades para prevenir, por exemplo, o acesso concorrente a estruturas de dados no programa que são compartilhadas entre as threads.

É imprescindível entender que um aplicativo possui uma área de disputa de memória compartilhada entre todas as threads, chamada Região Critica. Para que uma aplicação se torne confiável, ela terá que garantir que somente uma thread utilizará a Região Crítica por vez e o programador deve perceber quando mais de uma thread modifica uma determinada área de memória em comum.

Não é necessária a sincronização quando os valores da memória compartilhada não são modificados, ou seja, caso as threads só comparem ou tomem qualquer outro tipo de decisão que não venha a modificar os dados da memória compartilhada entre as threads. Para quem estudou um pouco de sincronismo de banco de dados, é mais fácil entender o sincronismo em java.

Para entender a utilização de threads programaticamente, podemos utilizar um exemplo! *Sabe quando você quer gerar um arquivo em PDF e, enquanto ele está sendo gerado, aparece uma barrinha de que vai enchendo à medida que o arquivo vai se concluindo?* Pois é, esses processos paralelos e simultâneos podem ser executados por meio de threads! Vamos pensar...

```
public class GeraPDF implements  Runnable {
  public void run () {
    // Código para gerar PDF.
  }
}

public class BarraDeProgresso implements  Runnable {
  public void run () {
    // Código da barra de progresso.
  }
}

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

    GeraPDF gerapdf = new GeraPDF();
    Thread threadDoPdf = new Thread(gerapdf);
    threadDoPdf.start();

    BarraDeProgresso barraDeProgresso = new BarraDeProgresso();
    Thread threadDaBarra = new Thread(barraDeProgresso);
    threadDaBarra.start();

  }
}
```

O código acima começa com duas classes – **GeraPDF**, que evidentemente gera o PDF; e **BarraDeProgresso**, que obviamente cria a barra de progresso! Observem que ambas implementam a interface **Runnable** – isso é necessário para que as instâncias dessa classe sejam executadas por uma thread. Aliás, essa interface possui um único método (**run()**), que cria a thread e a executa.

No método **main**, criamos os objetos de cada classe e passamos para a classe thread. *Como assim, professor?* Observem que primeiro geramos um objeto (**gerapdf**), em seguida criamos uma thread com esse objeto (**new Thread(gerapdf)**) e depois chamamos o método **start()**, responsável por chamar o método **run()** da classe específica – sem isso, não seria possível saber qual método **run()** deveria ser chamado.

4.8 JAVA: COLEÇÕES

Coleção é qualquer agregado de referências a objetos (elementos) em alguma estrutura de dados. As coleções são sempre de um mesmo tipo de elementos. Por exemplo, podemos ter uma coleção de Datas, uma coleção de Pessoas, ou uma coleção de Alunos. Em Java, os elementos de uma coleção devem ser sempre referências a objetos, não podendo ser tipos primitivos, como **int** e **char**.

Mas essa limitação na prática não existe, porque a cada tipo primitivo corresponde uma classe, como as classes Integer e Character. Por exemplo, para cada valor de **int** pode ser construído um objeto da classe Integer, que tem esse valor na sua única variável de instância. Podemos então ter coleções de Integer, de Double, de Character. *Bacana?*

Quando falamos em tipo dos elementos de uma coleção, estamos falando, portanto, do tipo das referências que podem ser armazenadas na coleção. Para facilitar, na prática usamos comumente expressões como "coleção de objetos" como uma forma abreviada de dizer "coleção de referências a objetos". Da mesma forma, dizemos, "coleção de Alunos" para "coleção de referências do tipo Aluno".

É um erro conceitual comum pensar que uma coleção de objetos armazena esses objetos dentro dela. O conceito de coleção é bem genérico, e significa qualquer agrupamento de objetos. Existem conceitos mais específicos para representar coleções com propriedades estruturais particulares. O mais simples é o conceito de uma lista. Outras formas conceituais são as ideias de conjunto, árvore, grafo, etc.

Vamos começar falando das Listas (**java.util.List**)! Ela se caracteriza por ser sequencial, com cada elemento ocupando uma posição relativa, indexada de **0** até **N-1**, onde **N** é a quantidade de elementos. Em uma lista, faz sentido falar em primeiro, segundo, terceiro elemento. Não há restrição em se colocar uma mesma referência em mais de uma posição da lista (i.e., a lista pode ter elementos duplicados).

Galera, não sei se algum de vocês já programou em C! *Se sim, vocês se lembram de como era complicado utilizar ponteiros, ponteiros para ponteiros, alocação dinâmica de memória?* Pois é, Java não exige que você faça nada disso! As coisas aqui são muito mais simples e já existe bastante funcionalidade implementada e pronta para ser utilizada. As principais classes implementadoras são: **ArrayList**, **LinkedList**, **Vector**.

```
ArrayList arrayList = new ArrayList();
LinkedList linkedList = new LinkedList();
Vector vector = new Vector();
```

```
//Observem que eu posso referenciar os objetos criados como List (Lista).

List list = new ArrayList();
List list = new LinkedList();
List list = new Vector();
```

MÉTODO	DESCRIÇÃO
add(Object)	Adiciona uma referência no final da lista e aceita referências de qualquer tipo;
add(int, Object)	Adiciona uma referência em uma determinada posição da lista;
size()	Informa a quantidade de elementos armazenado na lista;
clear()	Remove todos os elementos da lista;
contains(Object):	Verifica se um elemento está contido em uma lista;
remove(Object):	Retira a primeira ocorrência de um elemento de uma lista;
remove(int):	Retira elementos pela sua posição na lista;
get(int):	Recupera um elemento de uma determinada posição da lista;
indexOf(Object):	Descobre o índice da primeira ocorrência de um elemento;

Agora vamos falar um pouco sobre Conjuntos (java.util.Set). Trata-se de outra categoria de coleção, em que não há noção de posição relativa (primeiro, segundo, etc) e que não podem existir elementos duplicados. Um exemplo simples é o conjunto de países que estão na Comunidade Europeia. Não há interesse em se repetir países, e não há ordem relativa (um país não está antes ou depois de outro).

Como os conjuntos não possuem ordem, as operações baseadas em índice que existem nas listas não aparecem nos conjuntos. Set é a interface Java que define os métodos que um conjunto deve implementar. As principais classes implementadoras são: HashSet, TreeSet e LinkedHashSet. Cada implementação possui suas características sendo apropriadas para contextos diferentes.

Galera, sendo bem sincero, não vejo grandes vantagens no uso de sets. Tudo bem, algumas vezes, eles possuem performance melhor que a das listas, mas não é nada demais. O que eu acho importante é saber essas características básicas e diferentes para outros tipos de coleções. Vamos ver agora um exemplo e algumas das suas principais operações e suas descrições.

```
HashSet hashSet = new HashSet();
TreeSet treeSet = new TreeSet();
LinkedHashSet linkedHashSet = new LinkedHashSet();

//Observem que eu posso referenciar os objetos criados como Set (Conjunto).

Set set = new HashSet();
Set set = new TreeSet();
Set set = new LinkedHashSet();
```

MÉTODO	DESCRIÇÃO
size()	Retorna a quantidade de elementos armazenados no conjunto;
isEmpty	Verifica se o conjunto está vazio;
add()	Adiciona um elemento específico ao conjunto;
remove()	Remove um elemento específico do conjunto;

Agora vamos ver Filas (java.util.Queue). Trata-se de outra categoria de coleção, semelhante a filas, em que o primeiro elemento a entrar será o primeiro elemento a sair (FIFO). Basta lembrar da fila de um supermercado. A primeira pessoa que entrou na fila será a primeira pessoa a sair da fila. Aliás, ela possui operações de inserção, remoção e inspeção.

Galera, percebam que uma fila é uma lista, porém com um objetivo diferente! A fila é desenhada para ter elementos inseridos no final da fila e removidos no início da fila. As principais classes implementadoras da interface Set são: PriorityQueue e LikedList. Cada implementação possui suas características sendo apropriadas para contextos diferentes.

```
LinkedList linkedList = new LinkedList();
PriorityQueue priorityQueued = new PriorityQueue();

//Observem que eu posso referenciar os objetos criados como Queue (Fila).
```

```
Queue queue = new LinkedList();
Queue queue = new PriorityQueue();
```

MÉTODO	DESCRIÇÃO
add()	Insere um elemento na fila;
remove()	Remove um elemento da fila;
element()	Retorna o primeiro elemento da fila;

Por fim, vamos falar sobre os Mapas (java.util.Map). Aqui já começamos diferente, porque mapas não são coleções. Pensem comigo: muitas vezes queremos buscar rapidamente um objeto, dada alguma informação sobre ele (Ex: dada a placa do carro, obter todos os dados). Poderíamos utilizar uma lista e percorrer todos os elementos, mas isso é péssimo para a performance – aqui entra o mapa!

Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor. É equivalente ao conceito de dicionário, utilizado em várias linguagens. Algumas linguagens, como Perl ou PHP, possuem um suporte mais direto a mapas, onde são conhecidos como matrizes ou arrays associativos. O objetivo é mapear uma chave a um valor.

Em um dicionário eu associo um vocábulo a uma definição. A chave é um objeto utilizado para recuperar um valor. O mapa costuma aparecer junto com outras coleções, para poder realizar essas buscas. As principais classes implementadoras da interface Map são: HashMap, TreeMap e LinkedHashMap. Cada implementação possui suas características sendo apropriadas para contextos diferentes.

MÉTODO	DESCRIÇÃO
clear()	Remove todos os pares chave/valor do mapa;
containsKey(k)	Retorna true se o mapa invocador contiver o objeto k como chave;
containsValue(v)	Retorna true se o mapa contiver o objeto v como chave;
entrySet()	Retorna um conjunto que contenha as entradas no mapa;
equals()	Retorna true se mapas contiverem as mesmas entradas;
get()	Retorna o valor associado com a chave k;
remove(k)	Remove a entrada que tiver chave igual a k;

4.9 JAVA: STREAMS E SERIALIZAÇÃO

O que é Stream? Vamos ver um exemplo bacana: *sabem quando vocês veem um jogo de futebol ao vivo pela internet? Ou quando vocês assistem ao julgamento de um réu pelo website do Supremo Tribunal Federal?* Pois é, isso é streaming! Em outras palavras, trata-se de um fluxo de dados contínuo (nesse caso, vídeo) transmitido de uma fonte de dados para um destino específico.

Em Java, quando dizemos que um objeto é serializado, estamos querendo dizer que ele será transformado em bytes, e poderá ser armazenado em disco ou transmitido por um stream. Um stream é um objeto de transmissão de dados, em que um fluxo de dados serial é feito através de uma origem e de um destino. Os tipos mais comuns de stream são o FileOutputStream e o FileInputStream.

Ambos os streams são utilizados para manipular objetos serializados. O primeiro é um fluxo de arquivo que permite a gravação em disco; o segundo é um fluxo de arquivo que permite a leitura em disco – ou seja, o inverso! Galera, em suma, a serialização serve para salvar, gravar e capturar o estado de um objeto. Assim, posso criar um objeto, gravá-lo em um arquivo e futuramente utilizá-lo.

A serialização permite que eu transforme uma instância de um objeto em uma sequência bytes. É bacana porque, nesse formato, eu posso enviá-lo pela rede, salvar no disco ou comunicar uma JVM com outra. Eu congelo o estado atual de um objeto e, lá no destino, descongela-se esse estado sem perda de dados. O exemplo que eu costumo usar é jogo de videogame!

Sabe quando você já está lá quase zerando o jogo? E se você desligar o videogame e perder tudo? Pois é, é útil salvar o estado atual! *Como faço isso?* Você deve implementar a interface **Serializable** do pacote **java.io.Serializable**. *O que tem nessa interface, professor?* Nada, mas ela indica para a JVM que você deseja que determinada classe esteja habilitada para ser serializada. Não encontrei questões!

4.10 JAVA: CLASSES E OPERAÇÕES DE I/O

Professor, o que é I/O? É uma sigla para Input/Output! Em português, seria E/S – ou Entrada/Saída! Isso indica a comunicação entre sistemas de processamento de dados (Ex: Computador) e o mundo externo (Ex: Humano). Entradas são dados recebidos pelo sistema e Saídas são dados enviados pelo sistema. A melhor maneira de entender isso é por meio de um exemplo!

Uma operação de entrada de dados seria aquele dado lido de uma base de dados (Ex: **read()** – lê dados da entrada padrão do sistema). Já uma operação de saída de dados seria aquele dado escrito em uma base de dados (Ex: **write()** – escreve dados na saída padrão do sistema). Agora que vocês já sabem o que são operações de Entrada/Saída, podemos prosseguir!

O Java é capaz de processar arquivos que requisitam grandes quantidades de dados persistentes. *O que é isso, professor?* São aqueles dados armazenados dentro de arquivos – fala-se persistente porque ele dura além da finalização de um programa. Esse processamento de arquivos é uma capacidade que a linguagem oferece para ler e gravar dados! *Bacana?*

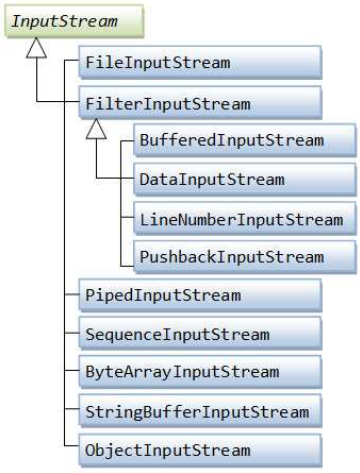
Pessoal, se vocês quiserem trabalhar com entrada e saída de dados em nossa linguagem, recomenda-se utilizar classes que estão dentro do pacote **java.io**. *Por que?* Porque lá vocês encontrarão classes que oferecem funcionalidades como manipulação de entrada/saída de bytes, manipulação de entrada/saída de caracteres, buffers de leitura e escrita, conversão de formatos, entre outros.

As classes abstratas **InputStream** e **OutputStream** definem, respectivamente, o comportamento padrão dos fluxos de entrada (para ler bytes) e dos fluxos de saída (para escrever bytes). Em outras palavras, essas duas classes abstratas permitem manipular a entrada e a saída de dados como uma sequência de bytes, sejam um arquivo, um **BLOB** de uma base de dados, uma conexão remota via sockets, etc.

Vamos falar um pouco sobre a Classe **InputStream**! Ela oferece a funcionalidade básica de leitura de um byte ou de uma sequência de bytes a partir de alguma fonte. Podemos utilizar o método **read()** – o valor de retorno desse método é um inteiro, que pode ser o byte lido do próprio método ou número de bytes lidos – quando o retorno for igual a **-1**, é informado que o final do arquivo foi atingindo.

```
InputStream is = new FileInputStream("arquivo.txt");
int valor = is.read();
```

Observem no exemplo acima que a classe abstrata **InputStream** foi declarada inicializando uma classe dependente **FileInputStream**. Essa classe recebe uma **String** como argumento do seu método construtor – ele representa a definição de onde está localizado o arquivo. *Entendido?* Podemos ver abaixo a hierarquia das classes que dependentes da Classe **InputStream**:

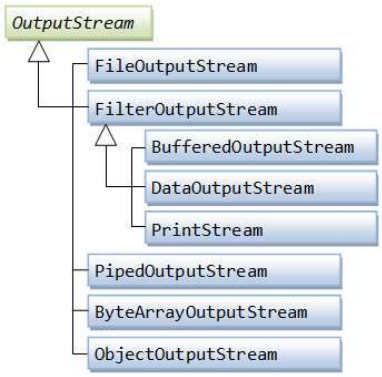


Abaixo podemos ver uma pequena descrição das principais classes:

Classe	Descrição
ByteArrayInputStream	Valores são originários de um arranjo de bytes.
FileInputStream	Bytes com originalidade de um arquivo.
FilterInputStream	Filtra os dados de um InputStream.
BufferedInputStream	Faz a leitura de grandes volumes de bytes que armazena em um buffer interno.
DataInputStream	Permite a leitura de representações binárias dos tipos primitivos de Java.
ObjectInputStream	Oferece o método readObject para a leitura de objetos que foram serializados para um ObjectOutputStream.
PipedInputStream	Faz a leitura de um pipe de bytes cuja origem está associada a um objeto PipedOutputStream.

Já a Classe OutputStream é responsável por transferir os bytes para algum destino. O método **write()** tem a função de escrever em forma de bytes para o destino em que enviará os dados. *Que destino é esse?* Bem, isso não importa! Quando o sistema precisar escrever em uma saída, basta ele chamar o método que utiliza a classe abstrata, visto que ele aceita qualquer dependente de OutputStream.
Observem um exemplo abaixo:

```
File arquivo = new File("Teste.txt");
FileWriter fw = new FileWriter(arquivo);
   fw.write("25");
   fw.flush();
```



Abaixo podemos ver uma pequena descrição das principais classes:

Classe	Descrição
FileOutputStream	Escreve em um arquivo ou em um descritor de arquivo.
FilterOutputStream	Filtra saídas, transformando dados ou provendo funcionalidades adicionais.
PipedOutputStream	Cria comunicações pipe ao se conectar com entradas pipe.
ByteArrayOutputStream	Implementa uma saída na qual dados são escritos em um array de bytes.
ObjectOutputStream	Escreve tipos de dados primitivos.

4.11 JAVA: CODE CONVENTIONS

Existe uma série de convenções de código para a linguagem de programação Java. *Para quê, professor?* Para padronizar a forma de um código-fonte em Java. Esse padrão é utilizado internamente no desenvolvimento dos códigos-fontes Java e não engloba o aspecto de arquitetura de sistema, i.e., não interfere na engenharia do software, focando apenas na forma de escrita e organização do código-fonte em si.

Segundo introdução do documento, os motivos principais para se manter um padrão de codificação em um trabalho em equipe é que 80% do tempo gasto no desenvolvimento de software é alocado para manutenção; ademais, dificilmente qualquer software será mantido a vida toda pelo mesmo autor; por fim, padrões aumentam a legibilidade do software, permitindo uma leitura mais clara e rápida.

O documento engloba padrões para: nomes de arquivos, classes, pacotes, variáveis, constantes, etc; organização dos arquivos; indentação; comentários; declarações de funções, variáveis e outras; disposição das expressões; política para espaços em branco e quebras de linha; entre outros. Vamos ver agora algumas das principais convenções:

PRINCIPAIS CONVENÇÕES
Todos os códigos-fonte devem obrigatoriamente possuir um comentário de Javadoc no início;
A primeira linha de código no arquivo de origem será a instrução de pacote seguido de quaisquer declarações de importação;
Os nomes dos pacotes devem começar com um nome de domínio de nível superior em letras minúsculas (por exemplo, com. ou edu);
Os nomes de classe e de interface devem ser substantivos e devem utilizar Camel Case com cada palavra sendo capitalizada (por exemplo, TesteDeSoftware);
Todos os arquivos de classe devem ter um Javadoc para classe;
Listar as variáveis da seguinte forma: variáveis estáticas, variáveis de instância (público, protegido, sem acesso especificado e privadas);
Listar os métodos da seguinte forma: construtores e, em seguida, os métodos (que devem ser agrupados por funcionalidade, não escopo);
Nomes de métodos devem ser verbos e devem utilizar Camel Case com cada palavra sendo capitalizada, a não ser que a primeira letra (Ex: getHoraUteis);
Nomes das variáveis devem ser verbos e devem utilizar Camel Case com a primeira letra minúscula (Ex: horasTrabalhadas);
As variáveis devem começar com alfabetos. Variáveis de um caractere (por exemplo, i, j, ou k) devem ser evitadas e usado apenas para variáveis temporárias;
Tente fazer todas as variáveis de classe não públicas e acessíveis apenas através de métodos;

Constantes devem ter todas as letras maiúsculas, com palavras separadas por um sublinhado (por exemplo MAX_HORAS_TRABALHADAS);
Tente usar valores numéricos como constantes (por exemplo, int MAX_HORAS_TRABALHADAS = 24);
Tente inicializar variáveis locais onde elas foram declaradas e evite linhas maiores que 80 caracteres;
Cada linha deve conter apenas uma instrução e declarações de if-else, for, while, do e switch devem sempre conter chaves {};

4.12 JAVA: JAVADOC

Você já sabe como escrever comentários usando o // e /* ... */. Estes comentários servem como documentação que é boa principalmente para o desenvolvedor que está escrevendo a classe e para os desenvolvedores que irão prosseguir os trabalhos sobre essa classe. Porém, há um outro tipo de documentação em Java. Este tipo é apropriado se você estiver escrevendo uma API que será usada por outras pessoas.

Você pode usar o programa Javadoc que vem com o JDK e pode ser encontrado no diretório **bin** de instalação do JDK. Por padrão, Javadoc gera arquivos HTML que descrevem pacotes e tipos. Cada arquivo HTML gerado descreve um pacote ou um tipo. Dentro da descrição de um tipo, você também pode descrever métodos e campos do tipo, além de construtores do tipo - se o tipo for uma classe.

O input do Javadoc é o código-fonte e o Javadoc é capaz até de verificar alguns erros de compilação no código. Isto significa que você pode gerar a documentação antes mesmo de o projeto estar completo. A saída, por padrão, é um conjunto de arquivos HTML, mas você pode personalizar o Javadoc para formatar a saída de uma forma diferente.

Comentários também podem conter *tags*, que são palavras-chave especiais que podem ser processadas. *Tags* devem aparecer após a descrição de um comentário. Pode haver múltiplas *tags* em um comentário e é possível ter *tags* com nenhuma descrição. Por exemplo: o seguinte comentário contém a tag @author que especifica o nome do desenvolvedor. Vejam outras *tags* que podem ser usadas:

```
/**
 * Isto é um comentário!
 * @author Diego Carvalho
 */
```

▪ @author
Isso indica que o autor ou autores do código.
▪ @version
Isto indica a versão da classe ou método. Usado para classes e interfaces.
▪ @ Param
Este descreve um parâmetro de método. Utilizado de métodos e construtores.
▪ @return
Este descreve o objeto de retorno de um método. Utilizado de métodos que não retornam void.
▪ @throws
Descreve uma exceção que pode ser acionada pelo método. Equivalente a @exception.

▪ @see
Isto indica uma referência, que pode ser um URL, um outro elemento na documentação, ou apenas algum texto.
▪ @serial
Isso indica se um campo é serializável.
▪ @deprecated
Isto indica que um método está obsoleto e não é um substituto.
▪ @since
Especifica que o componente sendo comentado é válido a partir de uma determinada versão.

4.13 NOVIDADES: JAVA 8

Pessoal, vocês já devem saber que nós temos uma nova versão! A cobrança ainda é raríssima (só encontrei uma questão), mas é bom saber um pouco sobre as principais novidades. A grande novidade foram as *Lambda Expressions*! Pois é, Java agora tem algumas características de programação funcional. *Professor, o que é exatamente programação funcional?*

É um paradigma de programação que trata a computação como uma avaliação de funções matemáticas e que evita estados ou dados mutáveis. Ela enfatiza a aplicação de funções, em contraste com a programação imperativa, que enfatiza mudanças no estado do programa. As Lambda Expressions permitem passar comportamentos, ou funções como argumentos em uma chamada de método.

Para entendê-las, é necessário conhecer os conceitos de funções de primeira classe e literais. Tradicionalmente no Java, um método (função, procedimento) somente existe como membro de uma classe. Isso significa que, embora você possa ter uma variável "apontando" para um objeto, você não pode guardar um método numa variável. *Bacana, pessoal?*

Tudo aquilo que é permitido referenciar numa linguagem (no caso, objetos ou tipos primitivos), passar como parâmetro para outras funções, etc, é dito ser "de primeira classe". Outras linguagens, entretanto, permitem que funções e outras coisas mais (como classes) sejam referenciados e passados como argumento. No JavaScript, por exemplo, é bastante comum passar uma função para uma variável.

O suporte a funções de primeira classe simplifica em muito a construção de certas funções. Aliás, elas vão um passo além, não só permitindo passar funções como parâmetro para outras funções, mas também permitindo que as mesmas sejam expressas como literais. *Professor, o que é um literal?* Um literal é uma notação que representa um valor fixo no código fonte.

Em outras palavras, através do uso da própria sintaxe você consegue criar um objeto que de outra forma exigiria a combinação de duas ou mais funcionalidades diferentes. No Java 8, o literal para uma expressão lambda consiste em uma lista de argumentos (zero ou mais) seguida do operador -> seguida de uma expressão que deve produzir um valor. Exemplos:

```
() -> 42          // Não recebe nada e sempre retorna "42"
x -> x*x          // Recebe algo e retorna seu quadrado
(x,y) -> x + y    // Recebe dois valores e retorna sua soma
```

Em suma, as expressões lambda busca concisão no código, i.e., fazer mais escrevendo menos. Com frequência, não há razão para se exigir que uma função esteja sempre acompanhada de uma classe, e o uso dessas expressões evita muitas construções desnecessárias. Evita-se criar classes anônimas com apenas um método para solucionar algum problema. Quem se interessar mais:

http://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html#use-case

ESSA LEI TODO MUNDO CONHECE:
PIRATARIA É CRIME.

Mas é sempre bom revisar o porquê e como você pode ser prejudicado com essa prática.

1. 

Professor investe seu tempo para elaborar os cursos e o site os coloca à venda.

2. 

Pirata divulga ilicitamente (grupos de rateio), utilizando-se do anonimato, nomes falsos ou laranjas (geralmente o pirata se anuncia como formador de "grupos solidários" de rateio que não visam lucro).

3. 

Pirata cria alunos fake praticando falsidade ideológica, comprando cursos do site em nome de pessoas aleatórias (usando nome, CPF, endereço e telefone de terceiros sem autorização).

4. 

Pirata compra, muitas vezes, clonando cartões de crédito (por vezes o sistema anti-fraude não consegue identificar o golpe a tempo).

5. 

Pirata fere os Termos de Uso, adultera as aulas e retira a identificação dos arquivos PDF (justamente porque a atividade é ilegal e ele não quer que seus fakes sejam identificados).

6. 

Pirata revende as aulas protegidas por direitos autorais, praticando concorrência desleal e em flagrante desrespeito à Lei de Direitos Autorais (Lei 9.610/98).

7. 

Concurseiro(a) desinformado participa de rateio, achando que nada disso está acontecendo e esperando se tornar servidor público para exigir o cumprimento das leis.

8. 

O professor que elaborou o curso não ganha nada, o site não recebe nada, e a pessoa que praticou todos os ilícitos anteriores (pirata) fica com o lucro.

Deixando de lado esse mar de sujeira, aproveitamos para agradecer a todos que adquirem os cursos honestamente e permitem que o site continue existindo.