1 – A Tecnologia Java

1.1 - Sobre

A tecnologia Java é considerada uma linguagem de programação e também uma plataforma.

1.2 - A Linguagem de programação Java

A Linguagem de programação Java é uma linguagem de alto nível que pode ser caracterizada por todos seguintes termos:

- Simples
- Orientada à Objetos
- Distribuída
- Multithreaded
- Dinâmica
- Arquitetura Neutra
- Portável
- Alto Desempenho
- Robusta
- Segura

Na Linguagem de programação Java, todo o código fonte é primeiro escrito em arquivos de texto simples (.java). Estes arquivos fonte são então compilados para arquivos com a extensão .class pelo compilador javac (java compiler). Um arquivo .class não contém código nativo de um processador, ao invés disso, ele contém bytecodes - a linguagem de máquina da “Máquina Virtual Java” (Java JVM). O lançador java então roda sua aplicação em uma instância da Máquina Virtual Java.

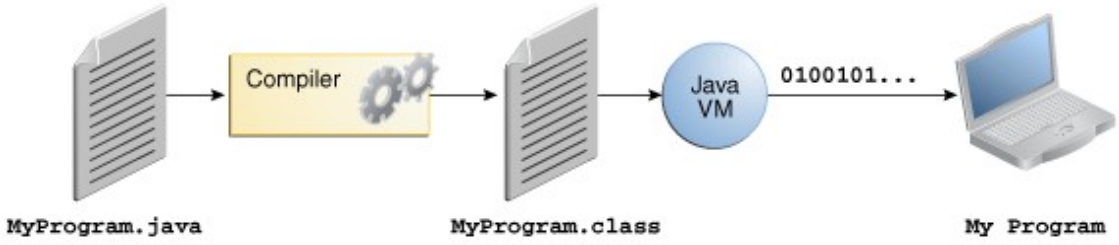


Fig. 1 - Visão geral do processo de desenvolvimento de software

Como a JVM está disponível para vários diferentes sistemas operacionais, o mesmo arquivo .class são capazes de rodar no Microsoft Windows, Solaris™ Operating System (Solaris OS), Linux ou OSX. Algumas máquinas virtuais realizam passos adicionais em tempo de execução para aumentar a performance da aplicação, por exemplo, encontrar gargalos de desempenho e recompilar para código nativo as seções de código mais utilizadas.

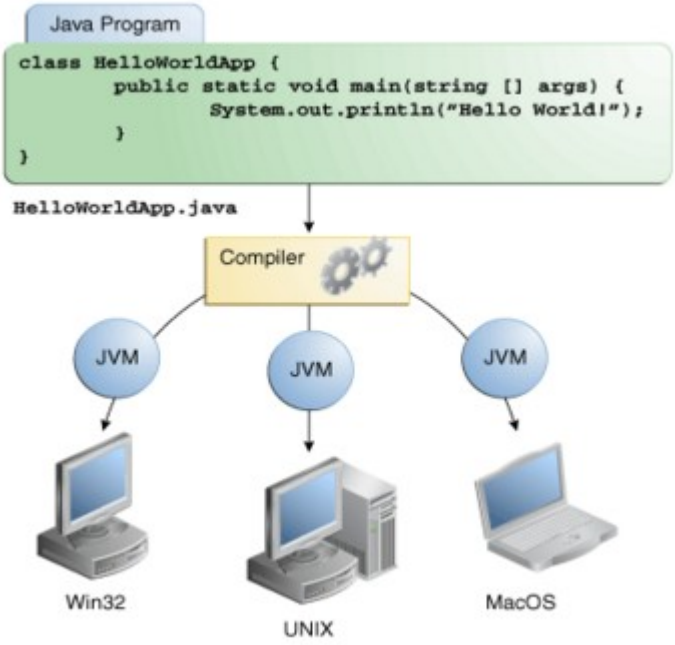


Fig. 2 - Por meio da JVM, a mesma aplicação é capaz de rodar em múltiplas plataformas

1.3 - A Plataforma Java

Uma plataforma é um ambiente de Hardware ou Software no qual um programa roda. Nós já mencionamos as plataformas mais populares como Microsoft Windows, Linux, Solaris OS e OSX. A maioria das plataformas pode ser descrita como uma combinação de sistemas operacionais e hardware subjacentes. A plataforma Java é diferente da maior parte das outras plataformas, pois é uma plataforma composta apenas de Software que roda em cima de outras plataformas compostas por hardware.

A plataforma Java possui dois componentes:

- A Máquina Virtual Java (*Java Virtual Machine*)
- A Interface de Programação de Aplicações Java (*Java Application Programming Interface - API*)

A API Java é uma extensa coleção de componentes de software que fornecem recursos úteis. Ela é agrupada em bibliotecas (libs) de classes e interfaces relacionadas; estas bibliotecas são conhecidas como pacotes (packages).

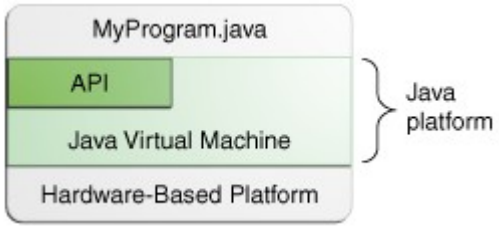


Fig. 3 - A *API* e a *Java Virtual Machine* isolam o programa do *hardware* subjacente.

1.4 - O que a Tecnologia Java pode fazer?

O propósito geral é fornecer uma linguagem de programação de alto nível e

uma poderosa plataforma de software. Qualquer implementação completa da plataforma Java deverá fornecer as seguintes funcionalidades:

- **Ferramentas de Desenvolvimento**: fornecem todas ferramentas necessárias para compilar, rodar, monitorar, "debugar" e documentar suas aplicações. Como um novo desenvolvedor, as principais ferramentas utilizadas serão javac (compiler), java (launcher) e javadoc (ferramenta para documentação de código).
- **Java Application Programming Interface (API)**: oferece um amplo conjunto de classes utilitárias prontas para uso em suas próprias aplicações. Ela engloba objetos básicos, itens de segurança e rede, geração de arquivos XML e acesso a bancos de dados e mais. A API Java é muito extensa, para obter uma visão geral do que ela contém, consulte o link: https://docs.oracle.com/javase/8/docs/index.html (Java Platform Standard Edition 8 Documentation).
- **Tecnologias para Implantação (Deployment Technologies)**: fornece mecanismos padronizados para implantar suas aplicações para usuários finais
- **Ferramentas para Criação de Interfaces para Usuário (GUIs)**: as ferramentas JavaFX, Swing e Java 2D tornam possível a criação de Interfaces gráficas sofisticadas
- **Bibliotecas de Integração**: fornece diversas bibliotecas para integração de software, habilitando a aplicação a interagir com bancos de dados e objetos remotos.

1.5 - Como a Tecnologia Java pode mudar a minha vida?

Não podemos prometer que você terá fama, fortuna ou mesmo um emprego aprendendo Java, mas é muito provável que, no mínimo, você fará os seus programas de uma maneira melhor e com menos esforço do que com outras linguagens. Nós acreditamos que a Tecnologia Java irá lhe ajudar da seguinte forma:

- **Aprender o básico rapidamente**: Embora Java seja uma poderosa linguagem de programação orientada à objetos, é fácil de aprender,

especialmente para programadores já familiarizados com C ou C++;

- **Escrever menos código**: Algumas comparações de métricas de software (contagem de classes, contagem de métodos, etc) sugerem que um programa escrito em Java pode ser até quatro vezes menor que o mesmo programa escrito em C++;
- **Escrever melhor o código**: Java encoraja boas práticas de codificação e o "garbage collector" ajuda a evitar vazamento de memória. Orientação à Objetos e API facilmente extensível permitem reuso de código existente, testado e menos propenso à introdução de bugs;

Alem disso, Java ajudará você a desenvolver programas mais rapidamente, evitar dependência de plataformas específicas, escrever o programa uma única vez e poder rodar em diversos Sistemas Operacionais (Write once, run anywhere), desenvolver software distribuído mais facilmente.

2 - "Hello World" Application

2.1 - Checklist 

Para escrever seu primeiro programa em Java você vai precisar:

- Java SE Development Kit 8 (JDK 8) instalado.
- Um editor de textos simples

2.2 - Criando Sua Primeira Aplicação

Sua primeira aplicação HelloWorldApp, irá simplesmente apresentar a mensagem "Hello World!" no console. Para criar este programa você irá.

- **Criar um arquivo fonte**: Um arquivo fonte contém código, escrito em Java, que você e outros programadores podem compreender. Você pode usar qualquer editor de testo para criar e editar arquivos fonte.

- **Compilar o arquivo fonte para um arquivo .class**: O compilador Java (javac) pega o arquivo fonte e traduz seu texto em instruções que a máquina virtual Java (JVM) pode entender. As instruções contidas neste arquivo .class são conhecidas como bytecodes.

- **Rodar o programa:** O lançador de aplicações Java (java) utiliza a JVM para rodar a aplicação.

2.3 - Criar um Arquivo Fonte

Inicie o seu editor de texto de preferência. Em um novo documento, digite o código a seguir:

```
/**
* A classe HelloWorldApp implementa uma aplicação que
* simplesmente imprime “Hello World!” na saída padrão.
*/
class HelloWorldApp {
        public static void main(String[] args) {
                System.out.println(“Hello World!”); //Apresenta a mensagem.
        }
}
```

CUIDADO AO DIGITAR

Nota: Digite todo o código, comandos e nomes de arquivos exatamente como mostrado acima. O compilador Java (javac) e o lançador (java) são “*case-sensitive*”, portanto você deve usar Maiúsculas e Minúsculas corretamente. HelloWorldApp não é o mesmo que helloworldapp

Salve o código em um arquivo com o nome HelloWorldApp.java.

2.4 - Compilar o arquivo fonte para um arquivo .class

Abra o terminal (Linux) ou Command (Windows). A Janela deve estar mostrando o diretório atual. Para compilar o arquivo fonte, navegue até o diretório

onde o arquivo foi criado e então digite o comando a seguir:

javac HelloWorldApp.java

Se nenhum erro for encontrado no código, o compilador deve gerar o arquivo bytecode HelloWorldApp.class. Na janela do prompt digite dir (Windows)/ ls -l (Linux) para listar os arquivos contidos no diretório. Os seguintes arquivos deverão estar presentes:

HelloWorldApp.java
HelloWorldApp.class

Agora que você tem um arquivo .class, você pode rodar seu programa.

2.5 - Rodar o Programa

No mesmo diretório digite o comando a seguir:

java - cp . HelloWorldApp

Você deverá visualizar o conteúdo abaixo na sua tela (Windows):

C:\>myapplication>**java -cp . HelloWorldApp**
Hello World!

C:\>myapplication>

Parabéns! O seu primeiro programa em Java funciona! =D

Agora que você viu sua aplicação “Hello World!” (e provavelmente compilou e rodou), você pode estar se perguntando como ela funciona. Aqui está o código

novamente:

```
class HelloWorldApp {
        public static void main(String[] args) {
                System.out.println(“Hello World!”); //Apresenta a mensagem
        }
}
```

O programa HelloWorldApp consiste em três componentes: Comentário de código fonte, definição da classe HelloWorldApp e o método principal (main method). A explicação a seguir irá fornecer o entendimento básico do código.

2.6 - Comentários de Código Fonte

Comentários são ignorados pelo compilador, mas são úteis para outros programadores. A Linguagem de Programação Java suporta três tipos de comentários:

/* texto */
O compilador ignora tudo a partir do /* até */.

/** documentação */
Indica comentário de documentação. O compilador ignora este tipo de comentário. A ferramenta de documentação java (javadoc) utiliza estes comentários para preparar automaticamente documentação de código.

// texto
O compilador ignora tudo a partir do // até o final da linha.

2.7 - Definição da Classe HelloWorldApp

A forma mais básica de declarar uma classe é:

```
class NomeDaClasse {
        . . .
}
```

A palavra chave class inicia a definição da classe acima, chamada NomeDaClasse. O código relativo a esta classe aparece entre as chaves { }.

2.8 - O Método Principal (main method)

Em Java, toda aplicação deve conter um método principal, cuja assinatura é:

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

Os modificadores public e static podem ser escritos em qualquer ordem, mas a convenção existente sugere o uso conforme acima. O parâmetro do método pode receber qualquer nome, mas a maioria dos programadores utiliza “args” ou “argv”.

O método principal é similar a função main em C e C++; ele é o ponto de entrada da aplicação e irá, subsequentemente, invocar todos os outros métodos requeridos pelo programa.

O método principal aceita um único argumento: um array de elementos do tipo String. Este parâmetro é o mecanismo através do qual você pode passar informações para sua aplicação em tempo de execução, por exemplo:

```
java MyApp arg1 arg2
```

Cada String no array é chamada de argumento de linha de comando (*command-line argument*). Estes argumentos permitem que o usuário altere o comportamento de uma aplicação sem a necessidade de recompilá-la. Por exemplo, em um programa de ordenação, o usuário poderia informar à aplicação se os dados serão ordenados em ordem crescente ou decrescente.

Finalmente temos:

System.out.println(“Hello World!”);

Utiliza a classe *System* da biblioteca Java, para imprimir a mensagem “Hello World!” na saída padrão do sistema.

3 - Variáveis Primitivas

Java é uma linguagem de programação estaticamente tipada, significa que todas as variáveis devem ser declaradas antes de serem usadas. Isto envolve informar o nome e o tipo de cada variável:

int numero = 1;

Desta forma, você está informando ao seu programa que um campo chamado “numero” existe, possui um dado numérico e tem um valor inicial “1”. O tipo de dado de uma variável determina os valores que ela pode conter, bem como das operações que podem ser executadas a partir dela. Além de “int”, a Linguagem de Programação Java suporta outros sete tipos de dados primitivos. Um tipo primitivo é pré-definido pela linguagem e é nomeado por uma palavra reservada. Valores primitivos não compartilham estado com outros valores primitivos. Os oito tipos de dados primitivos suportados pela Linguagem Java são:

- **byte**: O tipo de dado “byte” representa um número inteiro com sinal de 8 bits. Ele tem um valor mínimo de -128 e máximo de 127 e pode ser utilizado para economizar memória em grandes arrays, onde economia de memória realmente importa. Eles também podem ser utilizados no lugar de um “int” quando os limites deste é pequeno e conhecido.
- **short**: O tipo de dado “short” representa um número inteiro com sinal de 16 bits. Ele tem um valor mínimo de -32.768 e máximo de 32.767. Assim como “byte” você pode utilizá-los para economizar memória em grandes arrays.
- **int**: Representa um número inteiro com sinal de 32 bits. Seu valor mínimo é -2^{31} e máximo $2^{31} - 1$. A partir do Java SE 8, é possível utilizar “int” para representar inteiros não assinalados, com valor mínimo 0 e máximo $2^{32} - 1$.
- **long**: O tipo de dado “long” representa um valor inteiro assinalado de 64 bits.
- **float**: Representa valores decimais (floating point) de 32 bits. Os limites mínimo e máximo estão além do escopo deste curso. Este tipo de dado nunca deve ser utilizado para representar valores precisos, tais como moeda. Para isto, deve ser utilizado java.math.BigDecimal.
- **double**: Representa pontos flutuantes de dupla precisão com 64 bits. Os limites mínimo e máximo estão além do escopo deste curso. Para valores decimais, é geralmente a escolha padrão, no entanto, deve-se evitar o uso deste tipo de dado para valores precisos, tais como moeda.
- **boolean**: Representa apenas dois valores possíveis: true (verdadeiro) ou false (falso). Utilize este tipo de dado para flags simples que representam condições verdadeiras ou falsas.
- **char**: Representa um único caracter Unicode de 16 bits. Possui o valor mínimo ‘\u0000’ (ou 0) e máximo ‘\uffff’ (ou 65.535).

Além dos oito tipos primitivos listados acima, Java fornece um suporte especial para cadeias de caracteres (strings) através da classe java.lang.String. Colocando seus caracteres entre aspas (“ “), ela irá automaticamente criar um novo objeto String; por exemplo, String s = “esta é uma string”;

Objetos do tipo String são imutáveis, o que significa que, uma vez criados, seus valores não podem ser alterados. A classe “String” não é tecnicamente um tipo de dado primitivo, mas considerando seu suporte especial recebido pela Linguagem,

você tende a pensar deste jeito.

3.1 - Valores Padrão (Default Values)

Nem sempre é necessário atribuir um valor quando um campo é declarado. Campos que são declarados mas não inicializados, receberão um valor padrão pelo compilador. Em geral, este valor será zero ou null, dependendo do tipo de dado. No entanto, confiar nestes valores padrão não é uma boa prática de programação.
A tabela abaixo apresenta os valores padrões dos tipos de dados:

Tipo de Dado	Valor Padrão (para campos)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	‘\u0000’
String (ou qualquer objeto)	null
boolean	false

Nota: Variáveis locais têm um comportamento um pouco diferente. O compilador nunca associa um valor padrão para uma variável local. Se você não inicializar sua variável local quando ela for declarada, garanta que ela irá receber um valor antes de você tentar usá-la. Acessar uma variável local que não foi inicializada irá resultar em um erro em tempo de compilação.

3.2 - Literais

Você pode ter notado que a palavra chave new não é usada para inicializar

uma variável de um tipo primitivo. Tipos primitivos são tipos especiais dentro da linguagem. Eles não são objetos criados a partir de uma classe. Um literal é uma representação de um valor fixo no código fonte. Conforme demonstrado abaixo, um literal pode ser associado a uma variável de um tipo primitivo:

```
boolean result = true;
char capitalC = ‘C’;
byte b = 100;
short s = 10000;
int i = 100000;
```

3.3 – Exercícios

1 – Crie uma classe com o método main chamada Calculo:

- Adicione 3 variáveis primitivas do tipo int gastoA, gastoB e gastoC, com os seguintes valores: 10, 15 e 20, respectivamente;
- adicione uma variável numeroValores do tipo int com o valor 3;
- atribua o valor da soma das 3 variáveis a uma variável soma do tipo int;
- atribua a uma variável resultado tipo int a soma dividido por numeroValores;
- imprima na tela a variável resultado, o que apareceu?
- altere o valor de gastoB para 10, execute a classe novamente, o que apareceu?
- agora vamos alterar variável resultado para float, execute a classe, qual o resultado? Por quê?
- por último altere tipo da numeroValores para float, o que aconteceu? Por quê?-

2 - Em uma empresa, existem tabelas com o seu gasto mensal. Para fechar o balanço do primeiro trimestre, precisamos somar o gasto total. Sabendo que, em Janeiro, foram gastos R$ 15.000,00, em Fevereiro R$ 23.000,00 e em Março, R$ 17.000,00, faça um programa que calcule e imprima o gasto total no trimestre. Siga esses passos:

• Crie uma classe chamada BalancoTrimestral com um método main;

• declare uma variável chamada gastosJaneiro e inicialize-a com 15000;

• crie também as variaveis gastosFevereiro e gastosMarco e inicialize-as com 23000 e 17000, respectivamente;

• crie uma variável chamada gastosTrimestre e inicialize-a com a soma das outras 3 variáveis;

• imprima a variável gastosTrimestre;

• adicione código para imprimir a média mensal de gasto, criando uma variável mediaMensal junto com uma mensagem. Para isso, concatene a String com o valor, usando “Valor da média mensal = “ + mediaMensal.

3 - Imprima todos os números de 150 a 300.

4 - Imprima a soma de 1 até 1000;

5 - Imprima todos os múltiplos de 3, entre 1 e 100

6 - Imprima os fatoriais de 1 a 10:

- O fatorial de um número n é n * n-1 * n-2... até n = 1. Lembre-se de utilizar parênteses nos cálculos (precedência de operadores matemáticos)
- O fatorial de 1 é 1
- O fatorial de 2 é (1!) * 2 = 2
- O fatorial de 3 é (2!) * 3 = 6
- O fatorial de 4 é (3!) * 4 = 24

- Faça um for que inicie uma variável n como 1 e fatorial (resultado) como 1 e varia de 1 até 10, imprimindo cada resultado.
- Altere o código do exercício anterior, aumente a quantidade de números que terão fatoriais impressos, até 20, 30, 40. Em um determinado momento, o programa vai começar a apresentar respostas erradas. Por quê?
- Mude de int para long para ver alguma mudança

4 - Conceitos de Orientação à Objetos

4.1 - O que é um Objeto?

Objetos são a chave para o entendimento de tecnologia orientada a objetos. Se você olhar ao redor agora, encontrará vários exemplos de objetos do mundo real: seu cão, sua mesa, sua televisão, sua bicicleta, etc.

Objetos do mundo real compartilham duas características: todos eles tem **estado** e **comportamento**. Cães têm estado (nome, cor, raça) e comportamento (latir, farejar, abanar o rabo). Bicicletas também têm estado (marcha atual, velocidade atual, cor) e comportamento (trocar de marcha, frear). Identificar estados e comportamentos de objetos do mundo real é uma boa maneira de começar a pensar em termos de programação orientada à objetos.

Objetos de software são conceitualmente similares a objetos do mundo real: eles também consistem em estado e comportamento. Um objeto armazena o seu estado em campos (variáveis em algumas linguagens de programação) e expoem seu comportamento através de métodos (funções em determinadas linguagens).

Métodos operam no estado interno de um objeto e servem como o principal mecanismo para comunicação com outros objetos. Esconder o estado interno de um objeto e exigir que toda a interação seja executada através de métodos é um conceito conhecido como **Encapsulamento de Dados** - Um princípio fundamental de programação orientada à objetos.

4.2 - O que é uma Classe?

No mundo real, você encontrará frequentemente muitos objetos do mesmo tipo. Podem haver milhares de outras bicicletas no mundo, todas do mesmo modelo. Uma classe é um modelo (*blueprint*) a partir do qual, objetos são criados individualmente.

A classe a seguir é uma possível implementação de uma bicicleta:

```
class Bicicleta {
```

```
    int marcha = 1;
    int velocidade = 0;

    void trocaMarcha(int novoValor) {
        marcha = novoValor;
    }

    void aumentaVelocidade(int incremento) {
        velocidade = velocidade + incremento;
    }

    void diminuiVelocidade(int decremento) {
        velocidade = velocidade - incremento;
    }

    void apresentaEstado() {
        System.out.println("velocidade atual: " +
            velocidade + " marcha atual: " +
            marcha);
    }
}
```

A sintaxe da linguagem Java pode parecer nova para você, mas o modelo desta classe é baseado no exemplo anterior sobre objetos do tipo bicicleta. Os campos marcha e velocidade representam o estado do objeto e os métodos trocaMarcha, aumentaVelocidade e diminuiVelocidade definem sua interação com o mundo exterior.

Você pode ter observado que Bicicleta não contém um método principal. Isso é porque ela não é uma aplicação completa, é apenas um modelo para bicicletas que podem ser utilizadas na aplicação. A responsabilidade de criar e usar novos objetos Bicicleta pertence à alguma outra classe da aplicação.

Aqui está uma classe *BicicletaDemo* que cria dois objetos Bicicleta e invocam seus métodos:

```
class BicicletaDemo {
        public static void main(String[] args) {
                // Cria dois objetos
                // Bicicleta diferentes
                Bicicleta bike1 = new Bicicleta();
                Bicicleta bike2 = new Bicicleta();


                // invoca métodos nestes
                // dois objetos
                bike1.aumentaVelocidade(10);
                bike1.trocaMarcha(2);
                bike1.apresentaEstado();

                bike2.aumentaVelocidade(10);
                bike2.trocaMarcha(2);
                bike2.aumentaVelocidade(10);
                bike2.trocaMarcha(3);
                bike2.apresentaEstado();
        }
}
```

A saída desta aplicação deve imprimir a velocidade atual e a marcha atual de cada bicicleta:

```
velocidade atual: 10 marcha atual: 2
velocidade atual: 20 marcha atual: 3
```

5 - Convenções de nomenclatura

Por que devemos utilizar convenções de nomenclatura :

- 80% do custo de um software é gasto com manutenções;
- Dificilmente um código será mantido pelo seu criador original;
- Códigos bem escritos, bem descritivos, bem anotados aumentam a produtividade e facilitam seu entendimento;
- Se você vende seu código como produto, você deve ter certeza que qualquer produto seu segue o mesmo padrão;
- Convenções padronizam métodos de usabilidade com boas práticas baseado no

conhecimento e experiência de um corpo especializado na área;

Lembrando que, para as convenções funcionarem, todos do projeto devem segui-las.

Vamos a algumas convenções:

5.1 Package e Import Statements

Primeiro os pacotes e depois os importes. Ex.

```
package com.matera;

import com.matera.Collection;
import com.matera.ArrayList;
```

5.2 - Classes e Interfaces

Classes Java e interfaces seguem algumas regras de formatação, veja:

- Sem espaço entre um método e o parênteses e o nome do método "(" início de lista de parâmetros;
- Abertura da chaves "{" aparece no fim da mesma linha que foi declarado o código ;
- Fechamento da chaves "}" começa uma linha alinhada no conjunto do método a qual foi criada, exceto quando há códigos em parte em branco(vazio) ou nulo }"devendo aparecer imediatamente depois de aberto com "{"
- Métodos são sempre separados por uma linha em branco.

Ex.

```
class Exemplo extends Object {
      int ivar1;
      int ivar2;

      Exemplo (int i, int j) {
```

```
            ivar1 = i;
            ivar2 = j;
        }

        int metodoVazio() {}

    }
```

5.3 - Convenção para nomes

As convenções de nomenclatura tem como objetivo tornar os programas mais compreensíveis, tornando-os mais fáceis de ler. Eles podem também fornecer informações sobre a função do identificador, por exemplo, quer se trate de um pacote, constante, ou de classe que pode ser útil na compreensão do código.

Tipo de indentificador	**Regras de nome**	**Exemplos**
Packages	O prefixo do nome do pacote deve ser único, deve sempre ser escrito em letras minúsculas todo-ASCII e deve ser um dos nomes de domínio de nível superior, atualmente com, edu, gov, mil, net, org, códigos de duas letras identificando os países, tal como especificado na norma ISO 3166, 1981. Componentes subseqüentes do nome do pacote varia de acordo com uma organização próprias convenções de nomenclatura internos. Tais convenções podem especificar que certos componentes do nome do pacote pode haver divisão, departamento, projeto, máquina, ou nomes de login.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik
Classes	Os nomes de classe devem ser substantivos, em maiúsculas e minúsculas	class Raster; class ImageSprite;

	com a primeira letra de cada palavra interna em maiúscula. Tente manter seus nomes de classe simples e descritivo. Sempre evite palavras-ligadas , evite todas siglas e abreviaturas, seja semântico.	
Interfaces	Nomes de interfaces devem ser usadas com as primeiras letras em maiúsculas como nome de classes.	interface RasterDelegate; interface Storing;

Metodos	Métodos devem ser verbos, com a letra minúscula em primeiro lugar, com a primeira letra de cada palavra interna em maiúscula.	run();
		runFast();
		getBackground();

Variáveis	Os nomes de variáveis não deve começar com underscore _ ou sinal de dólar $ ou números, mesmo que ambos não são permitidos. Os nomes de variáveis devem ser curtos, mas significativo. A escolha de um nome variável deve ser mnemônico, isto é, concebidos para indicar ao observador casual a intenção da sua utilização. Um personagem nomes de variáveis devem ser evitadas, exceto para temporários "descartáveis" variáveis. Os nomes comuns para variáveis temporárias são i, j, k, m, n e para inteiros, c, d, e e para caracteres	int i; char c; float myWidth;

Constantes	Os nomes de variáveis declaradas constantes de classes e de constantes ANSI deve ser todo em letras maiúsculas com palavras separadas por sublinhados ("_").	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int

		GET_THE_CPU = 1;

5.4 - Sequência de criação de uma classe/interface

	Parte da Class/Interface	Notas
1	Class/interface comentários (/**...*/)	
2	declaração de classe ou interface	
3	Comentários Class/interface implementadas (/*...*/), se necessário	Qualquer comentário que não seja apropriado ser descrito no item 1, referente a classes implementadas/extendidas
4	Classe (static) variáveis	Primeiro as variáveis de classe públicas, em seguida, os protegidos, depois o nível de pacote(sem modificador de acesso) e então as privadas.
5	Instance variables	idem ao item 4
6	Construtores	
7	Metodos	os métodos devem ser agrupados pelo seu nível de funcionalidade

5.5 - Exercícios

1. Reescreva a classe que foi criada no exercícios de tipos primitivos seguindo essa convenção.

6 - Criando Classes

A palavra classe vem da taxonomia da biologia. Todos os seres vivos de uma classe tem uma série de atributos e comportamentos, mas não são iguais, podendo variar nos valores desses atributos e como realizam esses comportamentos.

Em Java uma classe é uma espécie de projeto de algo. Definindo seus atributos e comportamentos (métodos), alguns públicos e outros privados. Muito semelhante à Biologia não?

Para exemplificar vamos imaginar um sistema para controle de Conta Corrente que será utilizado por um banco. Logo percebemos que uma entidade importante para nosso sistema é a própria conta!

Bom, vamos ver algumas informações necessárias para qualquer conta:

- código da conta
- nome do dono
- saldo

Agora vamos ver quais comportamentos esperamos da conta. O que ela deve fazer:

- sacar um valor

- depositar um valor
- retornar o saldo
- retornar o nome do dono

Temos então um bom projeto de conta, mas por enquanto é apenas isso. não podemos acessar seus atributos ou pedir que faça algo, a não ser que seja um atributo ou método estático. O que temos não é uma conta e sim uma especificação, uma Classe!

Esta classe é utilizada para criar instâncias, contas propriamente ditas. A partir delas podemos usar seus valores e comportamentos. Tais instâncias são Objetos.

Apesar de definirmos na classe que toda conta tem que ter um saldo, código e nome do dono é nos seus objetos que haverá “espaço” para armazenar esses valores.

Segue o código Java de nossa classe Conta somente com seus atributos:

```
class Conta {
    int codigo;
    String nomeDono;
    double saldo;

    // Métodos...
}
```

Vamos criar alguns métodos. Primeiro o de “sacar” que tem como finalidade reduzir em X o saldo. Perceba que métodos em linguagens Orientadas a Objetos são muito semelhantes às funções de outras linguagens como C ou PHP.

```
class Conta {
    // Atributos...
    void sacar(double valor) {
        double novoSaldo = this.saldo - valor;
        this.saldo = novoSaldo;
    }
}
```

A palavra “void” indica que o método não possui retorno, semelhante a um procedimento. Ele também recebe um parâmetro chamado “valor” que será o valor a ser reduzido do saldo. Ao acessarmos um atributo da própria classe utilizamos a palavra reservada “this” seguida de ponto, apesar de não ser obrigatório neste caso.

Mas espere e se a conta não tiver saldo suficiente? Vamos alterar nosso método para que ele possa validar isso:

```
class Conta {
    // Atributos…
    void sacar(double valor) {
        if (valor <= this.saldo) {
            // podemos escrever em uma única linha
            this.saldo =  this.saldo - valor;
        }
    }
}
```

Agora sim!!! Hum... espera, não seria bom se ele me informasse se deu certo ou não? Para isso vamos mudar o retorno de nosso método de “void” para “boolean” e retornar “true” se sacou ou “false” se não deu certo. No final do curso veremos como fazer isto de uma forma melhor com tratamento de exceções.

```
class Conta {
     // Atributos…
     boolean sacar(double valor) {
          if (valor <= this.saldo) {
               this.saldo =  this.saldo - valor;
               return true;
          } else {
               return false;
          }
     }
}
```

Agora vamos criar o depositar, este sera muito semelhante:

```
class Conta {
     // ...restante da classe
     void depositar(double valor) {
          double novoSaldo = this.saldo + valor;
          this.saldo = novoSaldo;
     }
}
```

6.1 - Atributos e métodos estáticos

Estáticos são os métodos e atributos que tem em sua definição a palavra reservada “static”. Eles não pertencem a um objeto e sim a classe.

Um atributo estático pode ser acessado diretamente da classe ou através de um objeto, além disso, caso tenha seu valor alterado ele é alterado em todos os objetos já que é a mesma variável.

Imagine a conta defina um valor máximo de saldo, comum a tomas as contas. Quem define o valor máximo? A classe!!

```
static double valorMaximo = 9999.99;
```

Um método estático tem um principio muito semelhante e seria um comportamento da Conta e não de suas instancias... Sim é difícil no inicio.

6.2 - Constantes

Para facilitar no futuro vamos criar um método que imprime na tela algumas informações da conta como dono e saldo. Mas Antes precisamos saber qual o tipo de moeda nossas contas utilizam e para isso definiremos uma constante com a sigla da moeda, no caso “R$”.

No Java uma constante é definida com as palavras reservadas “static” que define que ela pertence à classe e não ao objeto, seguida de outra palavra reservada “final”, que por sua vez, define que o valor de tal variável é imutável ou seja uma vez setado não pode ser alterado.

Ainda temos que definir o seu tipo, que pode ser primitivo ou não, o seu nome e por ultimo a atribuição de valor.

Constantes são declaradas antes dos atributos:

```
class Conta {
    static final String SIGLA_MOEDA = "R$";

    // Atributos e métodos...
}
```

Alguns devem ter se perguntado, se somente a palavra “final” já torna a variável imutável, então porque utilizar a “static”. Isto ocorre pois um atributo “final” não estático é somente um atributo imutável do objeto que pode tanto ser inicializado na sua declaração quanto em algum construtor, veremos construtores mais para frente.

Já quando ele é “final” e “static” ele é imutável, pertence a classe e seu valor deve ser inicializado na sua definição, durante a codificação ou em blocos de inicialização estáticos.

Vamos ao método citado no inicio deste tópico:

```
class Conta {
    // ...restante da classe
    void imprimirDonoESaldo() {
        System.out.println(“A conta do “ + this.dono +
            “ tem o saldo de: ” + SIGLA_MOEDA + this.saldo);
    }
}
```

7 - Instanciando Objetos

Bom já temos uma classe, porém, para utilizar seus métodos e atributos (a não ser que sejam estáticos) precisamos de um objeto.

Para isso vamos utilizar uma classe “App” com método “main”. La criaremos uma variável que será uma referência de objeto, já que, ela não guardara o objeto em si, mas, somente o seu posicionamento na memória. Chamaremos a referência de “minhaConta”;

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

A criação do objeto é feita com a palavra reservada “new” seguida pelo nome da classe. A palavra “new” invoca um construtor, veremos isso mais pra frente.

Uma vez que o objeto tenha sido criado usamos o operador “=” para realizar a atribuição do endereço desse objeto na referência.

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

Podemos agora, através da referência, utilizar as variáveis declaradas na classe conta:

```
class App {
    public static void main(String[] args) {
        // É possivel usar somente uma linha
        Conta minhaConta = new Conta();
        minhaConta.dono = "João";
        minhaConta.saldo = 2000.0; // ponto separa casas decimais
        minhaConta.imprimirDonoESaldo();
    }
}
```

Definimos aqui que o dono da conta é o "João" e seu saldo é de R$2000,00. Logo em seguida utilizamos o método criado agora pouco para imprimir o dono e o saldo.

Agora vamos alterar o saldo da conta do João utilizando o método sacar criado anteriormente.

```
class App {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.dono = "João";
        minhaConta.saldo = 2000.0;

        minhaConta.imprimirDonoESaldo();
        minhaConta.sacar(200);
        minhaConta.imprimirDonoESaldo();
     }
}
```

7.1 - Transferência

Imagine agora que surgiu a necessidade de transferir dinheiro de uma conta para outra. Como fazer isso? Bom, vou mostrar de duas formas uma delas mais “procedural” e a outra Orientada a Objetos como deve ser.

Primeira forma, como não fazer. vamos criar um método estático na classe conta que recebera três parâmetros, conta de origem, conta de destino e valor transferido. esse método usara os métodos sacar e depositar das contas para isso.

```
class Conta {
        // ...restante da classe
        static boolean transferirEstatico(Conta contaOrigem, Conta contaDestino,
                double valor) {

                boolean conseguiuRetirar = contaOrigem.sacar(valor);

                if (conseguiuRetirar) {
                        return false;
                } else {
                        contaDestino.depositar(valor);
                        return true;
                }
        }
}
```

Apesar de funcionar muito bem este exemplo ele fere alguns princípios de OO e não deveria ser utilizado. Afinal se eu estou chamando ele a partir de uma conta ele poderia ser escrito da seguinte forma:

```
class Conta {
    // ...restante da classe
    boolean transferir(Conta contaDestino, double valor) {
        boolean conseguiuRetirar = this.sacar(valor);

        if (conseguiuRetirar) {
            return false;
        } else {
            contaDestino.depositar(valor);
            return true;
        }
    }
}
```

Apesar de pequena, existe uma diferença muito relevante entre os dois métodos.

Momento de reflexão, por que utilizamos os métodos sacar e depositar no lugar de retirar e atribuir o valor diretamente? Para evitar repetição de código e facilitar a manutenção do mesmo. Afinal quem sabe como sacar dinheiro é o método "sacar" e **somente** ele.

Vejamos a utilização dos dois:

```
class App {
    public static void main(String[] args) {
        Conta minhaConta = new Conta();
        minhaConta.dono = "João";
        minhaConta.saldo = 2000.0;
        Conta minhaOutraConta = new Conta();
        minhaOutraConta.dono = "Maria";
        minhaOutraConta.saldo = 1000.0;
        // 50 reais para a Maria através da classe
        Conta.transferirEstatico(minhaConta, minhaOutraConta, 50);
        // 50 reais para a Maria através do objeto
        minhaConta.transferirEstatico(minhaConta, minhaOutraConta, 50);
        // 50 reais para a Maria através do objeto utilizando segundo método
        minhaConta.transferir(minhaOutraConta, 50);
    }
}
```

Experimente ler cada uma das chamadas como se fosse um texto comum, a terceira forma é muito mais natural não é mesmo?

8 - Modificadores de acesso

Os modificadores de acesso são palavras reservadas do Java que alteram a visibilidade e acesso às classes e membros de classe (atributos, métodos e construtores). Vejamos abaixo a explicação sobre eles.

Modificador	**Classe**	**Pacote**	**Subclasse**	**Mundo**
public	Sim	Sim	Sim	Sim
protected	Sim	Sim	Sim	Não
sem modificador	Sim	Sim	Não	Não
private	Sim	Não	Não	Não

8.1 - public

Uma classe com o modificador public pode ser acessada de qualquer lugar. Já um método, atributo ou construtor é visível para qualquer entidade que possa visualizar a classe que pertence.

8.2 - private

Não pode ser utilizado em classes. Os membros de classe com esse modificador não podem ser acessados ou usados por nenhuma outra classe, nem por classes subclasses.

8.3 - protected

O modificador protected também não pode ser usado em classes. Torna o membro acessível às classes do mesmo pacote ou subclasses.

8.4 - Sem modificador:

As classes ou membros que receberem esse modificador são acessíveis somente por classes do mesmo pacote, na sua declaração não é definido nenhum tipo de modificador

9 – Construtores

Construtores como o próprio nome já diz definem formas de construir um objeto. Eles são invocados a partir da palavra reservada “new”.

Mas se não criamos nenhum construtor como conseguimos instanciar uma conta no capítulo 7? Toda classe vem com um construtor **default** que não recebe nenhum parâmetro e foi ele que usamos.

Sua assinatura consiste no nome da classe seguido de seus parâmetros. Muito parecido com a definição de um método, porém, sem um tipo de retorno. Lembre apesar da semelhança construtores não são métodos.

Voltando ao exemplo da Conta vamos criar um construtor que receba como parâmetro o nome do dono e o saldo inicial, de modo que, no exemplo anterior possamos criar o objeto e atribuir um dono e um saldo de uma única vez:

```
class Conta {
	// ... Atributos
	Conta(String nomeDono, double saldoInicial) {
		this.nomeDono = nomeDono;
		this.saldo = saldoInicial;
	}
```

```
	// Métodos ...
}
```

Repare que o parâmetro “nomeDono” tem o mesmo nome do atributo e isso não gera nenhuma problema, pois, quando invocamos o “nomeDono” a partir da palavra “this” o Java sabe que tem que utilizar o atributo caso contrário a variável local, no caso o parâmetro.

Geralmente declaramos construtores logo após os

Se tentarmos compilar nosso código encontraremos um erro pois uma vez que declaramos um construtor **explicito** o construtor default que utilizamos até o momento deixa de existir. Não é o nosso caso, porém, se for necessário continuar utilizando o construtor default basta torná-lo explicito.

```
class Conta {
	// ... Atributos
	Conta() {
	}

	Conta(String nomeDono, double saldoInicial) {
		this.nomeDono = nomeDono;
		this.saldo = saldoInicial;
	}

	// Métodos ...
}
```

Vamos alterar agora nossa classe App para utilizar nosso novo construtor.

```
class App {
	public static void main(String[] args) {
		Conta minhaConta = new Conta(“João”, 2000.0);
		Conta minhaOutraConta = new Conta(“Maria”, 1000.0);
		 // Restante do método main...
	 }
}
```

Construtores são muito importantes para programação OO e também podem ser utilizados para inicializar atributos marcados como “final”, exceto constantes.

10 - Blocos de inicialização

Existem dois tipos de blocos de inicialização: os estáticos e os de instância. O primeiro é executado quando a classe é carregada na JVM enquanto o segundo quando uma instância (objeto) é criada.

Para este exemplo utilizaremos uma classe qualquer que possui trâs atributos: “CONSTANTE”, “valorEstatico” e “valor”.

```
class MinhaClasse {
     static final int CONSTANTE;
     static int valorEstatico;
     int valor;

     // Bloco de inicialização estático
     static {
           CONSTANTE = 10;
           valorEstatico = 100;
     }

        // Bloco de inicialização de instância
     {
           valorEstatico = 100;
           valor = 1000;
     }
}
```

Como demonstrado, nos blocos de inicialização estáticos é possível atribuir valores para constantes e atributos estáticos. Já nos blocos de inicialização de instância é possível atribuir valores para atributos estáticos e “normais”.

11 - Tipos Enum

Um *enum* é um tipo de dado especial que permite que variáveis recebam um conjunto de constantes pré-definidas. A variável deve ser igual à um dos valores que foram pré-definidos para ela. Exemplos comuns incluem direções de uma bússola (valores NORTE, SUL, LESTE E OESTE) ou dias de uma semana.

Porque elas são constantes, os nomes de um enum devem estar em MAIÚSCULAS.

Em Java, você define um tipo enum usando a palavra chave enum. Por exemplo, se você fosse criar um tipo enum para os dias da semana, ele seria declarado desta forma:

```
public enum Day {
      SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
            THURSDAY, FRIDAY, SATURDAY;
}
```

Você deve utilizar tipos enum sempre que precisar representar um conjunto fixo de constantes. Isso inclui tipos enum naturais, como os planetas do sistema solar e outros conjuntos que você conhece todos os possíveis valores em tempo de compilação. Abaixo um exemplo de como podemos usar o enum *Day*, definido acima:

```
public class EnumTest {

	Day day;

	public EnumTest(Day day) {
		this.day = day;
	}

	public void tellItLikeItIs() {
		switch (day) {
			case MONDAY:
				System.out.print(“Mondays are bad.”);
				break;
			case FRIDAY:
				System.out.print(“Fridays are better.”);
				break;
			case SATURDAY:
			case SUNDAY:
				System.out.print(“Weekends are best.”);
				break;
			default:
				System.out.print(“Midweek days are ok”);
				break;
		}
	}

	public static void main(String[] args) {
		EnumTest firstDay = new EnumTest(Day.MONDAY);
		firstDay.tellItLikeItIs();
		EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
		thirdDay.tellItLikeItIs();
		EnumTest fifthDay = new EnumTest(Day.FRIDAY);
		fifthDay.tellItLikeItIs();
		EnumTest sixthDay = new EnumTest(Day.SATURDAY);
		sixthDay.tellItLikeItIs();
		EnumTest seventhDay = new EnumTest(Day.SUNDAY);
		seventhDay.tellItLikeItIs();
	}
}
```

A saída do programa deve ser:

```
Mondays are bad.
Midweek days are ok.
Fridays are better.
Weekends are best.
Weekends are best.
```

Tipos enum em Java são mais poderosos do que os seus equivalentes em outras linguagens. A declaração de um enum define uma classe. O corpo da classe de um enum pode incluir métodos e outros campos. O compilador automaticamente adiciona alguns métodos especiais quando cria um enum. Por exemplo, eles têm um método estático chamado values que retorna um array contendo todos os valores do enum na ordem em que foram declarados. Este método é comumente usado combinado com um for-each para iterar sobre todos os valores do tipo enum. Por exemplo, o código da classe "*Planeta*" abaixo, itera sobre todos os planetas do sistema solar.

```
for (Planet p : Planet.values()) {
     System.out.printf("Your weight on %s is %f%n", p,
           p.surfaceWeight(mass));
}
```

No exemplo a seguir, Planet é um tipo enum que representa os planetas do sistema solar. Eles são definidos com as propriedades constantes *mass* e *radius*.

Cada constante enum é declarada com valores para mass e radius. Estes valores são passados para o construtor quando a constante é criada. Java exige que as constantes sejam definidas primeiro, antes de qualquer campo ou método. Além disso, quando existem campos e métodos, a lista de constantes enum deve terminar com ponto e vírgula.

Nota: O construtor de um tipo enum deve ter acesso privado ou privado de pacote (default). Ele automaticamente cria as constantes que são definidas no começo do corpo do enum. Você não pode invocar um construtor enum.

```
public enum Planet {
      MERCURY (3.303e+23, 2.4397e6),
      VENUS (4.869e+24, 6.0518e6),
      EARTH (5.976e+24, 6.37814e6),
      MARS (6.421e+23, 3.3972e6),
      JUPITER (1.9e+27, 7.1492e7),
      SATURN (5.688e+26, 6.0268e7),
      URANUS (8.686e+25, 2.5559e7),
      NEPTUNE (1.024e+26, 2.4746e7);

      private final double mass; // in kilograms
      private final double radius; // in meters
      Planet(double mass, double radius) {
            this.mass = mass;
            this.radius = radius;
      }
      private double mass() { return mass; }
      private double radius() { return radius; }

      // universal gravitational constant (m3 kg-1 s-2)
      public static final double G = 6.67300E-11;

      double surfaceGravity() {
            return G * mass / (radius * radius);
      }
      double surfaceWeight(double otherMass) {
            return otherMass * surfaceGravity();
      }
      public static void main(String[] args) {
            if (args.length != 1) {
                  System.err.println("Usage: java Planet
                        <earth_weight>");
                  System.exit(-1);
            }
            double earthWeight = Double.parseDouble(args[0]);
```

```
                double mass = earthWeight/EARTH.surfaceGravity();
                for (Planet p : Planet.values())
                      System.out.printf("Your weight on %s is %f%n",
                            p, p.surfaceWeight(mass));
                }
          }
    }
```

Se você rodar *Planet.class* por linha de comando com um argumento de 175, você terá a seguinte saída:

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

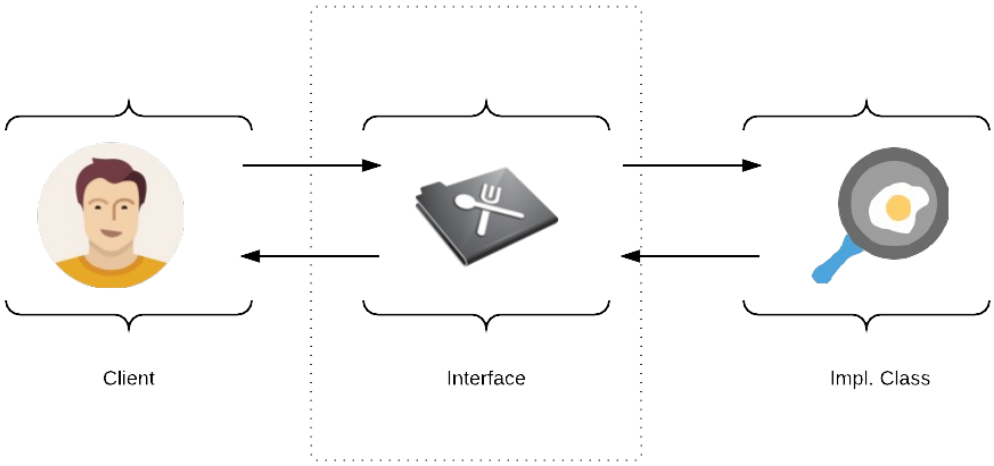
11.1 - Exercícios

1 - Crie um tipo enum para representar todos os meses do ano. O enum deverá conter um atributo inteiro chamado quantidadeDias cujo valor será a quantidade de dias de cada mês. Crie um programa que apresente o nome e o número de dias de cada mês do enum.

12 – Interfaces

Interface é um contrato onde a classe que a implementa deve obrigatoriamente obedecer. Uma abstração que pode ser feita com o a realidade, no

caso de um restaurante, é pensar que a interface é seu cardápio, lá estão descritos e definidos todos os pratos que o cliente pode pedir. O cliente pode por sua vez irá pedir um dos pratos existes e esperar receber exatamente oque foi solicitado, porém, eles não terão conhecimento e nem vão se importar em como o prato foi feito. Detalhes de como ele foi preparado, em java estarão codificados na classe de implementação da interface.



Vamos criar nosso menu e seu restaurante, definimos a interface chamada Menu e ele terá 3 pizzas: margherita, mozzarella e pepperoni.

```
public interface Menu {
	void margherita();
	void mozzarella();
	void pepperoni();
}
```

O Restaurante será chamado de Matera Pizza e irá implementar este menu através da classe de implementação MateraPizza:

```
public class MateraPizza implements Menu {

      @Override
      public void margherita() { /* bake margherita */ }

      @Override
      public void mozzarella() { /* bake mozzarella */ }

      @Override
      public void pepperoni() { /* bake pepperoni */ }

}
```

A interface **define os métodos**, mas nunca contém a implementação deles. Ela apenas **expõe o que o objeto deve fazer**, **e não como ele faz**, nem o que ele tem. Como ele faz é definido em uma implementação desta interface.

- Não possui implementação, apenas assinatura, ou seja, apenas a definição de seus métodos.
- Todos os métodos são implicitamente ***public*** e ***abstract***
- Não há como fazer uma instância de uma interface e nem criar um Construtor
- Seus atributos são implicitamente ***public, static*** e ***final***

13 - Herança

É um princípio da POO (Programação Orientada a Objetos), que permite a criação de novas classes a partir de outras previamente já criadas. Essas são chamadas de subclasses, ou classes derivadas; as classes que deram origem a elas, são chamadas de superclasses ou classes base.

Uma subclasse herda métodos e atributos de sua superclasse, apesar disto, é possível sobrescrevê-lo para uma forma mais específica de representar o comportamento do método herdado.

14 - Classes Abstratas

São classes que não serão instanciadas, ou seja, não serão c1riados objetos dela diretamente. Normalmente são referenciadas como superclasses abstratas.

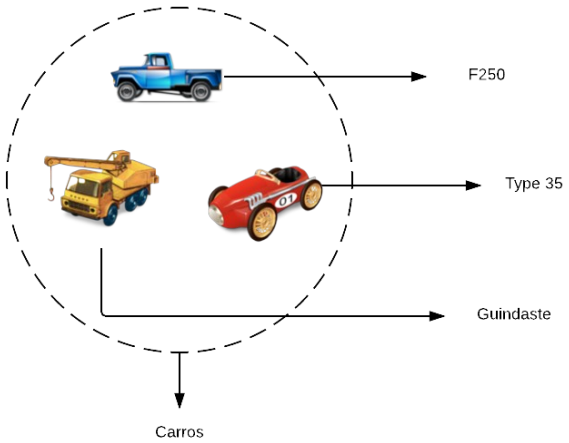
- É um tipo especial de classe e não podem ser criadas instancias dela.
- É usada para ser herdada, funciona como uma super classe.

Utilizadas para representar grupos que tem características comum, no entanto, que agem de maneiras diferentes em alguns pontos.

Exemplo 1: se você for o dono de uma empresa e for contratar alguém, você não contrata simplesmente um “Funcionário”, mas sim uma secretária, um técnico de TI, um gerente... enfim, você contrata alguém como uma especialidade para exercer uma função especifica.

Exemplo 2: você quando vai comprar um carro em uma concessionária não chega procurando por carro (abstrato), mas sim por um fiesta, gol, celta ...

No mundo real, existem apenas objetos, as classes são abstratas e servem para classificar grupos de objetos.



Elas definem apenas o conjunto de objetos com características semelhantes, elas

servem como modelos, que possuem apenas características gerais. Não é possível dizer qual a potência do motor de um “Carro”, nem o número de portas ou a média de consumo... características estas específicas de cada tipo de carro (F250, Guindaste...)

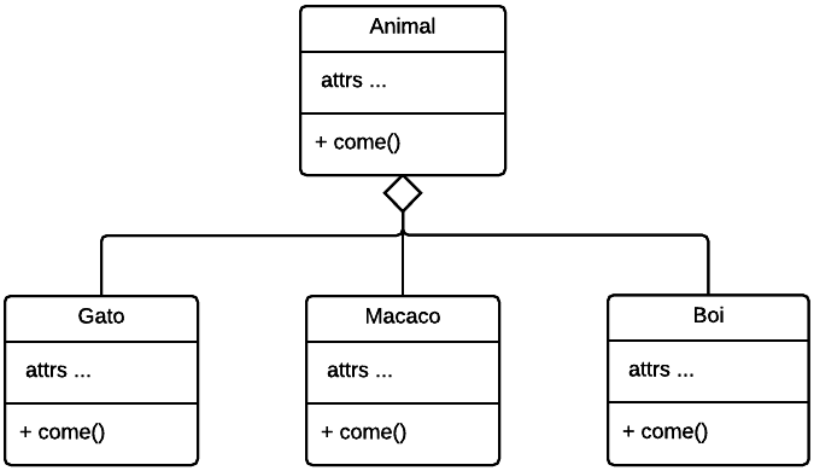
```
public abstract class Carro {

	public abstract int getVelocidadeMaxima();
	public abstract int getPortas();

	public void final acelerar() {
		...
	  }

	public void abastecer() {
		abasteceGasolina();
	 }

}

public class Fiesta extends Carro {
	@Override
	public int getVelocidadeMaxima() {
		return 200;
	  }

	@Override
	public int getPortas() {
		return 5;
	 }
}
```

```
public class Celta extends Carro {

      @Override
      public int getVelocidadeMaxima() {
           return 140;
     }

      @Override
      public int getPortas() {
           return 3;
       }

      @Override
      public void abastecer() {
           abasteceAlcool();
       }
}
```

15 - Polimorfismo

Polimorfismo significa “várias formas”, numa linguagem de programação, isso significa que podem existir várias formas de fazer “certa coisa”, que no caso, é a chamada de métodos.

Portanto, em JAVA, o polimorfismo se manifesta através da chamada de métodos, ou seja, ela pode ser executada de várias formas (polimorficamente), quem decide a forma a qual será executada é o objeto (instância) que recebe a chamada.




```
public class Animal {
	public void come() { System.out.println("ração"); }
}

public class Gato extends Animal { }

public class Macaco extends Animal {

	@Override
	public void come() { System.out.println("banana"); }
}

public class Boi extends Animal {

	@Override
	public void come() { System.out.println("capim"); }
}

Animal animal = new Animal();
animal.come() : _____

Animal animal = new Boi();
animal.come() : _____
```

```
Boi boi = new Boi();
boi.come() : _____________________________________________

Animal animal = new Gato();
animal.come() : _____________________________________________

Animal animal = new Macaco();
animal.come() : _____________________________________________

Gato gato = new Gato();
gato.come() : _____________________________________________

Macaco macaco = new Macaco();
macaco.come() : _____________________________________________
```

15.1 - Exercícios

1 - Escrever uma aplicação que declara uma variável do tipo OperacaoMatematica, e a partir de dados fornecidos pelo usuário, a aplicação deve realizar uma operação matemática e imprimir seu resultado.

- Oferecer para o usuário um menu (modo texto) para a escolha entre as operações disponíveis (soma, multiplicação, divisão e subtração).
- Aplicar o que aprendeu sobre polimorfismo.

2 - Dado uma empresa, todos funcionários dela ao final do ano recebem uma bonificação no valor de 50% de seu salário, com exceção do gerente que recebe 80% e coordenadores que recebem 65%. Dados os seguintes dados, desenhe o diagrama que representa sua hierarquia e faça sua implementação.

Cargo	Salário
Desenvolvedor	3000,00
Coordenador	5000,00
Gerente	8000,00
Secretária	2000,00

16 - Tratamento de Exceções

Em Java, os métodos dizem qual o contrato que eles devem seguir. Se, ao executar um método, ele não conseguir fazer o que deveria, ele precisa, ao menos, avisar ao usuário que o método não foi executado. Nessa situação podemos utilizar as Exceções.

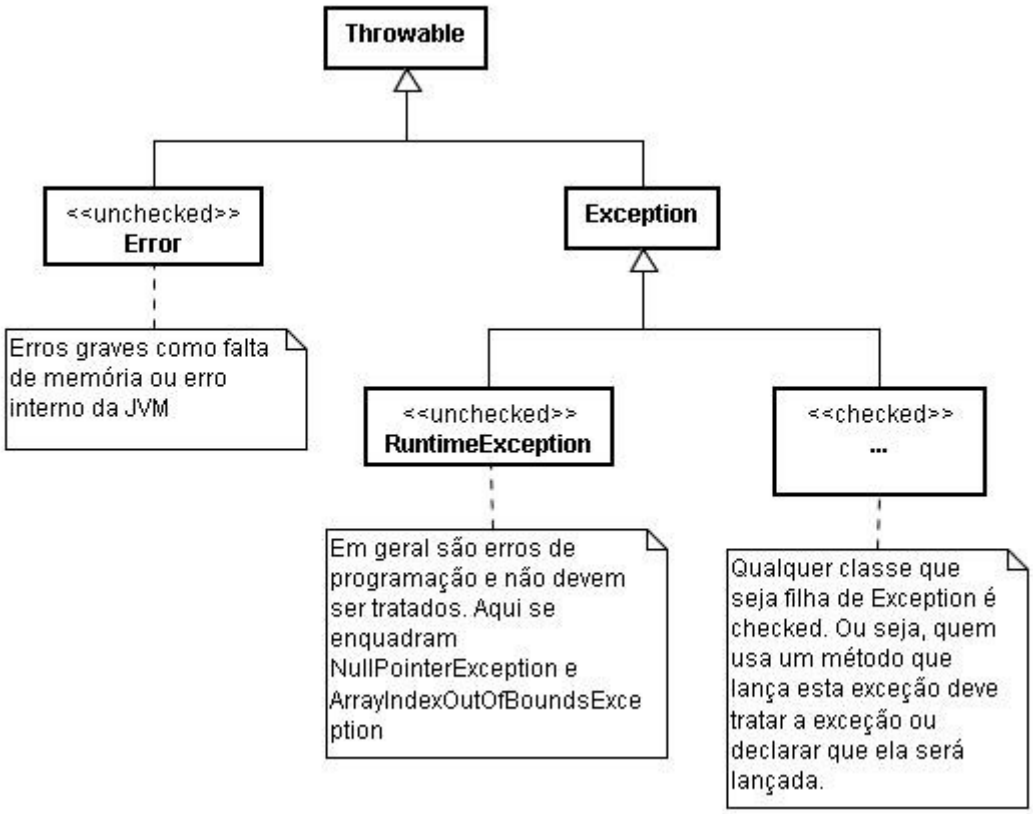
Existem dois tipos de exceções, que são:

- **Implícitas**: Exceções que não precisam de tratamento e demonstram serem contornáveis. Esse tipo origina-se da subclasse Error ou *RunTimeException*.
- **Explícitas**: Exceções que precisam ser tratadas e que apresentam condições incontornáveis. Esse tipo origina do modelo **throw** e necessita ser declarado pelos métodos. É originado da subclasse Exception ou *IOException*.

Existe também a formação de erros dos tipos **throwables** que são:

- **Checked Exception**: Erros que acontecem fora do controle do programa, mas que devem ser tratados pelo desenvolvedor para o programa funcionar.
- **Unchecked (Runtime)**: Erros que podem ser evitados se forem tratados e analisados pelo desenvolvedor. Caso haja um tratamento para esse tipo de erro, o programa acaba parando em tempo de execução (*Runtime*).
- **Error**: Usado pela JVM que serve para indicar se existe algum problema de recurso do programa, tornando a execução impossível de continuar.

16.1 - Hierarquia das Exceções



Existe uma diferença entre "**Erro (Error)**" e "**Exceção (Exception)**". O "Erro" é algo que não pode mais ser tratado, ao contrário da "Exceção" que trata seus erros, pois todas as subclasses de Exception (menos as subclasses RuntimeException) são exceções e devem ser tratadas. Os erros da classe Error ou RuntimeException são erros e não precisam de tratamento, por esse motivo é usado o **try/catch** e/ou propagação com **throw/throws**.

Palavras reservadas para o tratamento de exceções:

1. **catch**: Declara o bloco de código usado para tratar uma execução.
2. **finally**: Bloco de código, geralmente após uma instrução try-catch, que vai ser executado independente de que fluxo do programa for usado no tratamento de uma exceção.

3. **throw**: Usada para passar uma execução para o método que chamou o método atual.
4. **throws**: Indica o método que passará uma execução para o método que o chamou.
5. **try**: Bloco de código que será executado, podendo, porém, causar uma exceção.
6. **assert**: Avalia uma expressão condicional para verificar as alternativas do programador.

16.2 – Exercícios

Considere a seguinte classe:

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

	public static void metodoA(){
		System.out.println("entrou metodoA");
		int[] array = new int[5];
		for (int i = 0; i <= 7; i++) {
			array[i] = i;
			System.out.println(i);
		}
		System.out.println("saiu metodoA");
	}
}
```

1. Ao compilar e executar a classe o que acontece?
2. Se na chamada do metodoA() adicionarmos um try/catch oque será impresso na tela?

```
try{
      metodoA();
} catch (RuntimeException e) {
}
```

3. E se colocarmos o **try/catch** no loop **for**, o que será impresso na tela?

```
try{
      for (int i = 0; i <= 7; i++) {
            array[i] = i;
            System.out.println(i);
      }
} catch (RuntimeException e) {
}
```

4. Agora vamos adicionar um método finalizar() que necessariamente precisa ser executado antes que o programa finalize, como podemos garantir que esse método será executado?

5.Considere o seguinte metodo finalizar

```
public static void finalizar(){
      throw new RuntimeException("Erro nao checked");
}
```

Adicione uma chamada ao método finalizar no final do método no main da classe Inicio, o que acontece? Consegue compilar a classe?

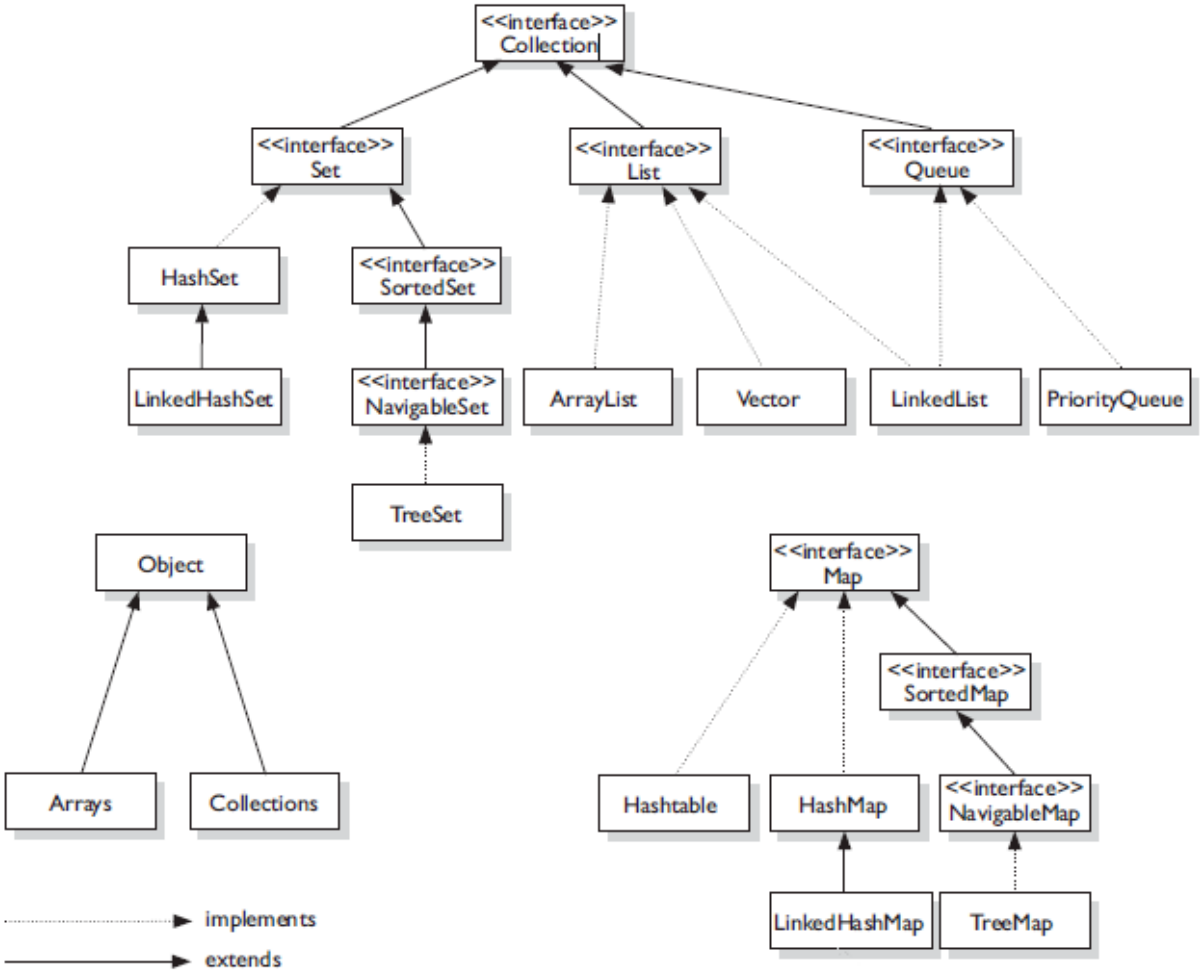
6. Agora altere o método finalizar para lançar uma exceção Exception, o que acontece? Consegue compilar? Por quê?

17 - Coleções

As coleções surgiram quando o comitê responsável pelo Java percebeu a falta

de um conjunto robusto de classes para suprir a necessidade de estruturas de dados básicas, como listas ligadas e tabelas de espalhamento. Com isso foi criado um conjunto de classes e interfaces conhecido como Collections Framework, que reside no pacote java.util desde o Java2 1.2. As Coleções em java visam facilitar a manipulação de Arrays em java.

Segue uma imagem da hierarquia de collections em java, suas interfaces e implementações.



Collection: O framework não possui implementação direta desta inferface, porém,

ela está no topo da hierarquia definindo operações que são comuns a todas as coleções;

Set: Está diretamente relacionada com a idéia de conjuntos. Assim como um conjunto, as classes que implementam esta interface não podem conter elementos repetidos. Usaremos implementações de SortedSet para situações onde desejarmos ordenar os elementos;

List: Também chamada de seqüência. É uma coleção ordenada, que ao contrário da inferface Set, pode conter valores duplicados. Além disso, temos controle total sobre a posição onde se encontra cada elemento de nossa coleção, podendo acessar cada um deles pelo índice.

Queue: Normalmente utilizamos esta interface quando queremos uma coleção do tipo FIFO (*First-In-First-Out*), também conhecida como fila.

Map: Vamos utilizá-la quando desejarmos uma relação de chave-valor entre os elementos. Cada chave pode conter apenas um único valor associado. Usaremos SortedMap para situações onde desejarmos ordenar os elementos.

Abaixo segue uma tabela para melhor entendimento de cada implementação das coleções;

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
Hashtable	x			No	No
TreeMap	x			Sorted	By *natural order* or custom comparison rules
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	By *natural order* or custom comparison rules
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	By to-do order

17.1 – Exercícios

Considere as seguintes classes TesteCollections e Pessoa

```
package com.matera;

import java.util.Iterator;
import java.util.Set;
import java.util.HashSet;

public class TesteCollections {
	public static void main(String[] args) {
		Set<Pessoa> array = new HashSet<Pessoa>();
		Pessoa pessoa = new Pessoa();
		pessoa.setId(new Long(2));
		pessoa.setNome("Joao");
		array.add(pessoa);
```

```
		pessoa = new Pessoa();
		pessoa.setId(new Long(1));
		pessoa.setNome("Marcos");
		array.add(pessoa);

		pessoa = new Pessoa();
		pessoa.setId(new Long(3));
		pessoa.setNome("Andre");
		array.add(pessoa);
		for (Iterator<Pessoa> iterator = array.iterator();
			iterator.hasNext();) {

			pessoa = iterator.next();
			System.out.println(pessoa.getId() + " - " +
				pessoa.getNome());
		}
	}
}

package com.matera;

public class Pessoa {
	private Long id;
	private String nome;

	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	public String getNome() {
		return nome;
	}
```

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

1 - Qual ordem os objetos pessoas foram impessos?
2 - Agora altere a implementação do HashSet para TreeSet, tente compilar, o que acontece?
3 - Implemente a interface Comparable<Pessoa> na classe pessoa e implemente o metodo compareTo comparando os id. Compile e execute TesteCollections novamente, o que monstrou na tela?

```
public class Pessoa implements Comparable<Pessoa>{
	….
	@Override
	public int compareTo(Pessoa o) {
		return this.id.compareTo(o.getId());
	}
}
```

4 - Se adcionarmos um o seguinte código, logo acima do loop for

```
pessoa = new Pessoa();
pessoa.setId(new Long(2));
pessoa.setNome("Paulo");
array.add(pessoa);
```

Ao compilarmos e executarmos a classe TesteCollections, o que é impresso? Por quê?

5 - Altere o medoto compareTo para comparar o nome. Compile e execute a classe TesteCollections, qual a ordem impressa, por quê?

6- Agora teste com as implementações de Map e List, fazendo as alterações necessárias