

Introdução à Linguagem Java

por Christian Cleber Masdeval Braz



Sumário

1 – Introdução ao Java	4
1.1 – Histórico da Linguagem	4
1.2 – Características da Linguagem	4
1.3 – Plataformas Java	5
1.4 – Java 2 System Development Kit	5
1.5 – A Máquina Virtual Java	5
1.6 – Garbage Collection	6
1.7 – Produtos e Terminologias da Tecnologia Java	7
2 – Estrutura básica de um programa	8
2.1 – Princípios Básicos da Linguagem	8
2.2 – Membros de uma Classe	8
2.3 – O método Main	9
2.4 – Modificadores de Acesso	9
2.5 – Convenção de Nomes	9
2.6 – Compilação e Execução de Programas	10
2.7 – Passagem de Parâmetro na Linha de Comando	10
3 – Variáveis e Operadores	13
3.1 – Nome de Variáveis	13
3.2 – Tipo de Dados	14
3.3 – Escopo de Variáveis	17
3.4 – Conversões Entre Tipos Primitivos de Dados	17
3.5 – Operadores	18
3.6 – Constantes	20
4 – Controle de Fluxo	22
4.1 – Comando if – else	22
4.2 – Comando switch	23
4.3 – Comando while	23
4.4 – Comando do - while	24
4.5 – Comando for	24
4.6 – Comando break	24
4.7 – Comando continue	25
5 – Métodos	28
5.1 – Modificadores de Acesso	28
5.2 – Nome de Métodos	28
5.3 – Argumentos	28
5.4 – Retornando Valor a Partir de um Método	29
5.5 – Passando Parâmetros para um Método	29

6 – Classes e Objetos em Java	32
6.1 – Encapsulamento	35
6.2 – Criando Objetos e Acessando Dados Encapsulados	36
6.3 – Construtores	37
6.4 – O Ponteiro <i>this</i>	39
7 – Mais Sobre Classes e Objetos	43
7.1 – Pacotes	43
7.2 – Atributos e Métodos de Classe	44
7.3 – Destrutores e o Método <code>finalize()</code>	45
8 – Herança e Polimorfismo	48
8.1 – Herança	48
8.2 – Polimorfismo	49
8.3 - Criação de um objeto de uma subclasse e o ponteiro <i>super</i>	52
8.4 – Métodos e Classes <code>final</code>	53
9 – Estruturando o Código com Classes Abstratas e Interfaces	56
9.1 – Classes Abstratas	56
9.2 – Interfaces	57
10 – Usando Strings e Arrays	62
10.1 – Strings	62
10.1.1 – Operações em Strings	63
10.1.2 – Convertendo primitivos para Strings e vice-versa	64
10.1.3 – A classe <code>StringBuffer</code>	64
10.2 – Arrays	65
10.2.1 – Arrays de Primitivos	65
10.2.2 – Arrays de Referências	66
10.2.3 – Arrays como Objetos	66
10.2.4 – Arrays e Exceções	67
10.2.5 – Arrays Multidimensionais	67
11 – Tratamento de Exceções	72
11.1 – Classes de Exceções	72
11.2 – Tratando de Exceções	73
11.2.1 – Capturando Exceções	73
11.2.2 – Deixando uma exceção passar através do método	75
11.2.3 – Capturando uma exceção e disparando outra diferente	76

1 – Introdução ao Java

A linguagem Java foi desenvolvida pela *Sun Microsystems* em 1995. Apesar de relativamente nova, a linguagem obteve uma espetacular aceitação por programadores do mundo inteiro, tendo se difundido como nunca antes ocorreu com uma linguagem de programação. Um fator que colaborou com isso, é o fato da linguagem possuir vantagens agregadas tais como: orientação a objetos, independência de plataforma, multitarefa, robusta, segura e distribuída. Com o advento da Internet, que tornou ainda mais necessário a comunicação entre plataformas heterogêneas, estes fatores fizeram com que o Java fosse a tecnologia perfeita para este novo cenário.

1.1 - Histórico da Linguagem

No início da década de 90, um pequeno grupo de projeto da *Sun* pretendia criar uma nova geração de computadores portáteis inteligentes, que pudessem se comunicar entre si de diversas formas. Para isso, decidiu-se criar uma plataforma de desenvolvimento onde o software pudesse ser executado em diversos tipos de equipamentos. Para o desenvolvimento desta plataforma foi escolhida a linguagem de programação C++.

A linguagem C++ não permitia realizar com facilidade tudo o que o grupo pretendia. Neste ponto, James Gosling, coordenador do projeto, decidiu criar uma nova linguagem de programação que pudesse atendê-los em suas necessidades.

A criação dos chips inteligentes foi abandonada devido ao alto custo de produção. Posteriormente, a explosão da Internet no mundo todo fez surgir a necessidade de uma tecnologia onde computadores de diferentes plataformas pudessem conversar. Surge daí, baseada na linguagem criada para o projeto de Gosling, a linguagem Java.

1.2 – Características da Linguagem

- Orientada a Objetos : Paradigma atual mais utilizado na construção de softwares. Dentre suas vantagens, podemos citar reaproveitamento de código e aumento da manutenibilidade dos sistemas assim desenvolvidos.
- Simples e Robusta : Java representa em muitos aspectos um aperfeiçoamento da linguagem C++. Ela possui certas características que permitem a criação de programas de forma mais rápida, pois tiram do programador a possibilidade de cometer erros que são comuns de ocorrer em C++. Algumas dessas características são o tratamento obrigatório de exceções e o gerenciamento automático de memória.
- Gerenciamento Automático de Memória : Em Java não existem ponteiros, isto é, não é permitido ao programador acessar explicitamente uma posição de memória. Java automaticamente gerencia o processo de alocação e liberação de memória, ficando o programador livre desta atividade. O mecanismo responsável pela liberação de memória que não está mais sendo utilizada é conhecido como *Garbage Collector*.
- Independência de Plataforma : Um dos elementos chave da linguagem Java é a independência de plataforma. Um programa Java escrito em uma plataforma pode ser utilizado em uma outra distinta da original. Este aspecto da linguagem é geralmente referenciado como “*write once, run anywhere*”. Isto é conseguido através da utilização da *Java Virtual Machine* (JVM) a qual roda numa plataforma específica e interpreta um programa Java para código de máquina específico da plataforma em questão. Como os programas em Java executam sob o controle da JVM, eles podem rodar em qualquer plataforma que possua uma disponível.

- Multi-threading : Um programa Java pode conter múltiplas threads para realizar várias tarefas em paralelo.

1.3 – Plataformas JAVA

A tecnologia Java está organizada em três plataformas com objetivos específicos:

- **Java 2 Standard Edition (J2SE):** ferramentas e APIs (Application Program Interface) essenciais para qualquer aplicação Java (inclusive para as outras plataformas). É suficiente a utilizarmos se quisermos desenvolver aplicações desktop com ou sem interface gráfica.
- **Java 2 Enterprise Edition (J2EE):** ferramentas e APIs para o desenvolvimento de aplicações distribuídas. Engloba tecnologias tais como RMI, EJB, CORBA, JMS, etc.
- **Java 2 Micro Edition (J2ME):** ferramentas e APIs para o desenvolvimento de aplicações para aparelhos portáteis (palms, celulares, eletrodomésticos).

Neste curso, será suficiente para nós utilizarmos apenas a plataforma J2SE, já que nosso objetivo principal é aprender a linguagem Java.

1.4 – Java 2 System Development Kit (J2SDK)

O J2SDK corresponde ao produto disponibilizado pela SUN que implementa a plataforma J2SE, provendo o ambiente básico necessário para o desenvolvimento de aplicações. Ele pode ser obtido no site da SUN no endereço <http://java.sun.com>.

O J2SDK consiste de:

- JRE (Java Runtime Environment): ambiente para **execução** de aplicações
- Ferramentas para desenvolvimento: compilador, debugger, gerador de documentação, empacotador JAR, etc
- Conjunto de APIs e código fonte das classes

1.5 – A Máquina Virtual Java

O JRE é um conjunto de programas que possibilita executar aplicações Java. O coração do JRE é a Máquina Virtual Java ou *Java Virtual Machine* (JVM). É a JVM que possibilita uma das características mais impressionantes da linguagem Java, a portabilidade do código. Vamos compreender um pouco como isso funciona

- No processo de compilação, ao invés do programa ser compilado para código de máquina da plataforma que vai ser executado, o programa é compilado para *bytecode*
- O *bytecode* é genérico, isto é, não é específico para nenhum sistema operacional em particular
- Quando um programa Java é executado, o arquivo *bytecode* é interpretado pelo interpretador da tecnologia java, que é denominado *Java Virtual Machine*. Existe uma JVM diferente para cada plataforma onde a tecnologia Java pode ser executada e deverá existir uma instalada no computador no qual será executado um programa Java. Os browsers, por exemplo, incorporam uma JVM para a execução de *applets*.

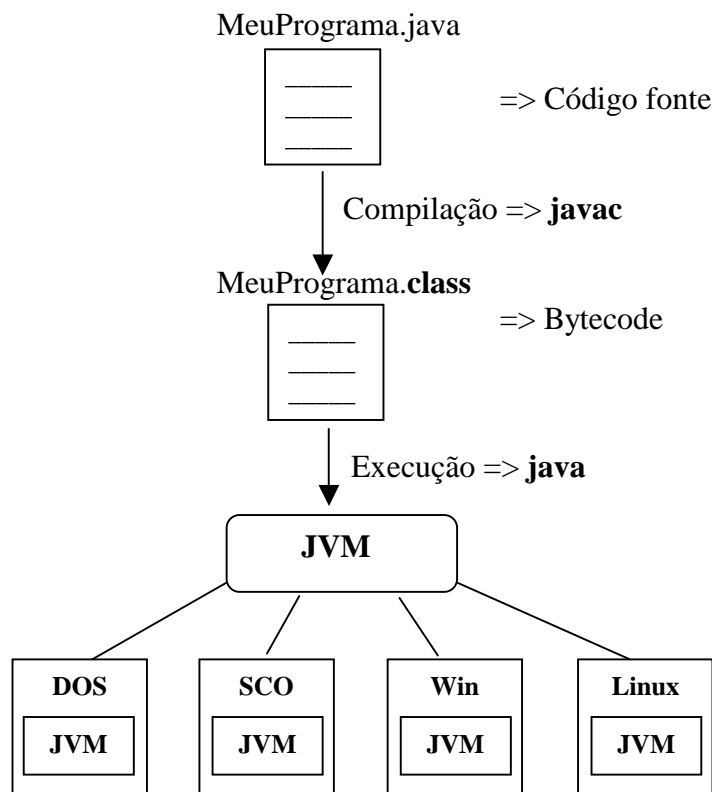


Figura 1- Esquema de compilação e execução de um programa em Java.

1.6 – Garbage Collection

No C e C++ (e em outras linguagens) o programa desenvolvido é responsável pela alocação e desalocação da memória, o que é um dos principais pontos causadores de erros nestes programas. Em Java, quando um objeto é criado (e quase tudo em Java é um objeto), a memória necessária é alocada automaticamente para ele, pois não há forma de se manipular a memória diretamente. Podem existir inúmeras variáveis no programa referenciando um mesmo objeto e, durante o ciclo de execução do programa, o Java verifica se um objeto ainda está sendo referenciado por alguma variável e, caso não esteja, o Java libera automaticamente esta área que não está sendo utilizada. Ou seja, quando não resta mais nenhuma referência para um objeto, ele é marcado para ser coletado pelo *garbage collector* ou “coletor de lixo”, que libera a memória ocupada por ele.

O coletor de lixo é executado de tempos em tempos num processo de baixa prioridade. Quando a JVM não está realizando nenhum processamento, ela executa o coletor de lixo que vasculha a memória em busca de algum objeto criado e não referenciado.

Quando uma grande quantidade de objetos ou objetos muito grandes não são mais necessários e não se quer esperar até que o coletor de lixo seja executado para liberar essa memória, pode-se chama-lo explicitamente no programa da seguinte forma :

```
System.gc();
```

1.7 – Produtos e Terminologias da Tecnologia Java

Produto	Descrição
Java Virtual Machine (JVM)	Interpreta arquivos .class (bytecode) para instruções específicas de plataforma.
Java Runtime Environment (JRE)	Somente o JVM não é suficiente para uma aplicação Java ser executada. O JRE consiste do JVM mais núcleo de classes da plataforma Java e arquivos de suporte. O JRE é o que deve estar instalado para a execução de uma aplicação Java.
Application Program Interface (API)	Biblioteca de classes e interfaces que auxiliam no desenvolvimento das aplicações.
Java Developer`s Kit (JDK)	Corresponde ao nome da tecnologia Java até Novembro de 1999. JDK 1.1 foi a última versão. É composto por : JVM, JRE, compilador, ferramentas e APIs.
Java 2 Platform, J2SE (Standard Edition)	É o nome da tecnologia a partir de Novembro de 1999. Corresponde ao JDK 1.2 na antiga terminologia. É composto por : JVM, JRE, compilador, ferramentas e APIs.
Javadoc	Corresponde a documentação das APIs. A última versão pode ser encontrada em http://java.sun.com/j2se/1.4.2/docs/api/ . A ferramenta Javadoc auxilia na criação da documentação das classes.

2 – Estrutura básica de um programa

O desenvolvimento de aplicações Java sempre é feito através de classes. A definição de uma classe Java deve respeitar a seguinte sintaxe :

```
<modificador de acesso> class <nome da classe>
{
    <Declaração das Variáveis de Instância (Atributos)>
    <Declaração de Métodos>
    public static void main( String args[] )
    {
        //corpo principal do programa
    }
}
```

Exemplo 1 :

```
┌───► Modificador de Acesso
public class Exemplo1 ──► Nome da Classe
{
    ┌───► Variável de Instância
    String mensagem = "Meu primeiro programa em Java!";

    public void Imprime_Msg()
    {
        //Mostra um texto na tela
        System.out.println(mensagem);
    }
    ┌───► Definição de Métodos

    public static void main (String [] args)
    {
        Imprime_Msg();
    }
    ┌───► Corpo do Programa
}
}
```

2.1 – Princípios Básicos da Linguagem

- Java é case-sensitive
- As classes, métodos ou blocos de código **sempre** estarão delimitados por um **abrir** ({) e **fechar** (}) de chaves.
- Um comando deve sempre ser finalizado por um **ponto e vírgula** (;)
- Dois tipos de comentário : //Comentário de uma linha e /* Comentário de Várias */
- Nomes de variáveis, classes e métodos devem sempre começar por **letras**, \$ ou _

2.2 – Membros de uma Classe

As variáveis de instância formam os atributos de um objeto e, juntamente com os métodos, são os elementos básicos para a formação de uma classe. Eles são denominados membros da classe.

Os atributos são espaços em memória reservados para armazenar informações durante a execução da classe. Eles constituem o **estado interno** de um objeto. Os atributos são inicializados no início da execução da classe e ficam disponíveis para utilização por

todos os seus métodos. Os métodos por sua vez definem as operações que podem ser realizadas pela classe. Mais sobre atributos será visto no capítulo Variáveis e Operadores e os métodos serão vistos em maiores detalhes no capítulo Métodos.

2.3 – O Método Main

Uma aplicação em Java é caracterizada por possuir o método **main()**. A declaração do método deve ser : **public static void main(String[] args)**.

O método main é um método especial pois representa o ponto de entrada para a execução de um programa em Java. Quando um programa é executado, o interpretador chamará primeiramente o método main da classe. É ele quem controla o fluxo de execução do programa e executa qualquer outro método necessário para a funcionalidade da aplicação.

Nem toda classe terá um método main. Uma classe que não possui um método main não pode ser “executada” pois não representa um programa em Java. Ela será sim, utilizada como classe utilitária para a construção de outras classes ou mesmo de um programa.

2.4 – Modificadores de Acesso

A declaração de uma classe geralmente começa com o modificador de acesso public, o que significa que pode ser acessada por qualquer outra classe. Se for omitido, a classe será visível apenas por outras classes do mesmo pacote que a contém.

2.5 – Convenção de Nomes

- Nome de Arquivos :
 - O código fonte de programas Java é armazenado em arquivos com extensão **.java**.
 - Um arquivo fonte poderá armazenar a definição de mais de uma classe, mas é uma boa prática armazenar apenas uma classe por arquivo fonte.
 - Use o nome da classe principal do arquivo como sendo o nome do arquivo.
 - Quando compilamos um arquivo fonte, será gerado um arquivo com extensão **.class** para cada classe definida e com o mesmo nome da classe. O arquivo **.class** contém o bytecode gerado com a compilação.
 - Exemplos : Cliente.java , ItemEstoque.java
- Nome de Classes : Utilize substantivos ou frases substantivas descritivas para nome de classes. Deixe maiúscula a primeira letra de cada substantivo que compõe o nome, por exemplo : MyFirstClassName
- Nome de Métodos : Use verbos para nome de métodos. Faça a primeira letra do nome minúscula com cada letra inicial interna maiúscula. Por exemplo : getUsername()
- Nome de Variáveis : Escolha nomes que indiquem o uso pretendido da variável. Utilize a primeira letra do nome minúscula e a inicial das outras palavras que formam o nome maiúscula. Exemplos : customerName, customerCreditLimit

2.6 – Compilação e Execução de Programas

Para compilarmos e executarmos um programa Java é necessário que a máquina possua o J2SE instalado.

Para compilarmos um programa na linha de comando, fazemos:

```
prompt> javac Exemplo1.java
- Saída do Compilador –
prompt>
```

Se não ocorrer nenhum erro de compilação após a execução do comando, será gerado o arquivo **Exemplo1.class**. Para o executarmos fazemos :

```
prompt> java Exemplo1 → Perceba que omitti-se a extensão .class
Meu primeiro programa em Java!
prompt>
```

Isto irá chamar a JVM que carregará o arquivo **Exemplo1.class** e tentará chamar seu método **main ()**. Se o **Exemplo1.class** chama métodos em outras classes, a JVM carregará estas outras classes somente quando for necessário.

2.7 - Passagem de Parâmetros na Linha de Comando

Aplicações Java permitem que se passe parâmetros através da linha de comando. Os parâmetros são separados por espaços em branco. Para passar um parâmetro que contém ele próprio espaços em branco deve-se colocá-lo entre aspas duplas.

Os parâmetros são passados para as aplicações através do vetor de strings do método **main**. Através do método **length** pode-se verificar o número de parâmetros passados. O acesso é feito indicando-se a posição no array, sempre iniciando em 0.

Exemplo 2 :

```
class ParametroLinhaComando
{
    public static void main( String [] args )
    {
        System.out.println("Meu nome é : " + args[0] + "e tenho " + args[1] + "
anos.");
        System.out.println("Tamanho do segundo parametro : " + args[1].length() +
" Número de parametros : " + args.length);
    }
}
```

A seguinte chamada a este programa

```
prompt> java ParametroLinhaComando Maria 18
```

gerará a seguinte saída :

```
prompt> Meu nome é Maria e tenho 18 anos. Tamanho do segundo
parametro : 2 Número de parametros : 2
```

Exercícios :

1 – Siga os passos abaixo para criar e executar sua primeira aplicação Java. Ela possui alguns erros do jeito que está. Tente descobrir e corrigi-los antes de compilar. Caso não consiga, compile e tente descobrir os erros a partir da saída apresentada.

a) Abrir um editor de textos e escrever o programa

```
class PrimeiraApp
{
    public void main( String args )
    {
        System.out.println("Primeira aplicação.")
    }
}
```

b) Salvar o arquivo dando a ele o mesmo nome da classe e com a extensão “**.java**”.

c) Compilar o arquivo

```
prompt> javac PrimeiraApp.java
```

Se a compilação foi com sucesso, ela gerou um arquivo “PrimeiraApp.class”, que é o arquivo a ser executado.

d) Execute o programa utilizando o interpretador java

```
prompt> java PrimeiraApp
```

2 – Examine o código abaixo e responda as questões que seguem.

```
public class HelloWorld {
    private int age;
    private String telephone;
    private String address;
    public static void main (String[] args){
        showMessage("Hello World!");
    }
    static void showMessage(String msg) {
        int messageNumber = 0;
        int messagePriority =;
        messageNumber = messageNumber + 1;
        System.out.println(msg + “ ” + messageNumber);
    }
}
```

- a) Qual o nome da classe ?
- b) Quais os nomes dos métodos ?
- c) Quais os nomes dos atributos ?
- d) Quais os nomes das variáveis do método ?

3 – Diga o que está de errado no código abaixo :

```
class mais_erros
{
    int 1_Idade;
    int segundaIdade;

    public void somaIdades (){
        System.out.println("A soma das idades é:" + (1_Idade + SegundaIdade) );

    public static void main (String[] args)
    {
        // Esta chamada de metodo ira imprimir a soma das idades na tela.
        SomaIdades;
    }
}
```

3 – Variáveis e Operadores

Uma variável é um nome simbólico para uma “porção de memória” na qual um valor pode ser armazenado. Toda variável tem as seguintes características :

- Deve ser declarada antes de ser utilizada;
- Possui um tipo, um identificador e um escopo;
- Podem ser locais, quando declaradas dentro de métodos, ou atributos, quando declaradas no corpo da classe;
- Podem ser inicializadas;

Como Java é uma linguagem fortemente tipada, **todas as variáveis devem ser declaradas antes de serem usadas**. O tipo de uma variável determina o tipo de informação que pode ser armazenada nela. Variáveis em Java podem ser declaradas como **atributos**, no corpo da classe, ou podem ser declaradas **localmente** em qualquer parte da implementação de um método. Variáveis também possuem um **escopo**, o qual determina onde no programa ela estará visível e poderá ser acessada.

A declaração de variáveis no Java segue a seguinte sintaxe :

tipo identificador [= valor] [, identificador [= valor]];

```
String msg = “Exemplos de
declaração de variáveis ”;
int umInteiro, UmInteiro;
float umFloat = 0.5;
char caracter = ‘i’;
boolean flag = false;
umInteiro = 90;
```

3.1 – Nome de Variáveis

O nome de uma variável deve começar com uma letra (a-z/A-Z), um underscore (_), ou um sinal de dólar (\$). Os caracteres subsequentes podem incluir os dígitos de 0 a 9. Como não existe restrição quanto ao tamanho dos nomes das variáveis, é recomendado que se escolha nomes significativos e que representem a real intenção de uso para a variável ou constante. Resumindo :

- Convenção : Escolha nomes que indiquem o uso pretendido da variável. Utilize a primeira letra do nome minúscula e a inicial das outras palavras que formam o nome maiúscula. Exemplos : customerName, customerCreditLimit

Exemplos de nomes para variáveis :

Legais ✓

a	item_Cost
itemCost	_itemCost
item\$Cost	itemCost2

Ilegais ✗

item#Cost	item-Cost
item*Cost	sem%
2itemCost	

Palavras-Chaves

Toda linguagem tem um grupo de palavras que o compilador reserva para seu próprio uso. Essas palavras-chaves não podem ser usadas como identificadores em seus programas. Em Java, as palavras reservadas são :

abstract	do	implements	package	throw
boolean	double	import	private	throws
break	else	*inner	protected	transient
byte	extends	instanceof	public	try
case	final	int	*rest	*var
*cast	finally	interface	return	void
*catch	float	long	short	volatile
char	for	native	static	while
class	*future	new	super	
*const	generic	null	switch	
continue	*goto	operator	synchronized	
default	if	*outer	this	

*os itens marcados com asterisco são para implementações futuras da linguagem.

3.2 – Tipos de Dados

Uma variável em Java pode ser de um dos quatro tipos de dados : classes, interfaces, arrays e tipos primitivos. No momento da compilação, o compilador classifica cada variável como uma **referência** ou um **valor primitivo**. Uma referência é um ponteiro para algum objeto. As classes, interfaces e os arrays são armazenados como referências. Os tipos primitivos são diretamente armazenados e seu tamanho é fixo. O tamanho e as características de cada tipo primitivo são consistentes entre as plataformas, não há variações dependentes de máquina como em C e C++, um benefício da portabilidade do Java.

Tipos Primitivos de Dados

Java possui 8 tipos primitivos onde 6 são tipos numéricos, um tipo *char* para caracteres e um tipo *booleano*.

Categoria	Tipo	Tamanho
Inteiro	byte	8 bits
Inteiro	short	16 bits
Inteiro	int	32 bits
Inteiro	long	64 bits
Ponto Flutuante	float	32 bits
Ponto Flutuante	double	64 bits
Caracter	char	16 bits
Lógico	boolean	true / false

- **Inteiro**

Os tipos de dados dessa categoria armazenam valores numéricos inteiros, isto é, números sem casas decimais. O Java possui 4 tipos primitivos desta categoria e o que diferencia um do outro é a sua capacidade de armazenamento.

Tipo	Tamanho	Variação
byte	8 bits	-128 a + 127
short	16 bits	-32.768 a +32.767
int	32 bits	-2.147.483.648 a +2.147.483.647
long	64 bits	9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

Por default, **literais**¹ inteiros são de 32 bits (int) e podem ser especificados como decimais, octais e hexadecimais. Para definir um literal como um número decimal, basta atribuir um número decimal qualquer, como octal deve-se acrescentar um 0 no início do número e como hexadecimal um 0x ou 0X. Os octais contêm os dígitos de 0-7 e os hexadecimais os caracteres 0-9 e A-F em maiúsculo ou minúsculo. É possível também definir literais inteiros do tipo long, ou seja, com 64 bits. Para isso, basta acrescentar um L no final do número. O efeito disso é que, se a variável foi definida como int, o compilador passa a interpretá-la como long.

Exemplos de literais do tipo Inteiro :

Decimal	0	1998	-23456
Octal	02	077	0123
Hexadecimal	0x0	0x2a	0X1FF
Long	365L	077L	0x1ffL

- **Ponto Flutuante**

Os tipos de dados de ponto flutuante armazenam números que possuem uma parte decimal, isto é, representam números reais. Em Java são dois os tipos dessa categoria : **float** e **double**. O tipo float permite representar valores reais com precisão simples (32 bits) enquanto o tipo *double* permite representar com precisão dupla (64 bits).

Deve ser utilizado o **ponto** como separador de casas decimais. Quando necessário, os números podem ser escritos em notação exponencial, usando o caracter “e” ou “E. Por exemplo : $4.2 * 10^6$ pode ser representado por 4.2e6.

A representação de um literal com ponto flutuante é, por default, considerada do tipo double. Para representar um literal do tipo float com precisão simples (32 bits) devemos incluir a letra F ou f no final do número.

Alguns exemplos de literais com ponto flutuante são : 1.0 , 4.7 , .47 , 1.22e19 ($1.22 * 10^{19}$), 4.6E-9 ($4.6 * 10^{-9}$), 6.2f , 6.21F

- **Caracter**

O único tipo dessa categoria é o tipo **char**. Uma variável do tipo char permite armazenar apenas um caracter. Caso seja necessário armazenar palavras ou textos deverá ser utilizado o tipo String, que não é um tipo primitivo. É necessário utilizar aspas simples para atribuir um literal para uma variável do tipo char, por exemplo : char letra = ‘a’;

¹ Enquanto variáveis armazenam valores, os literais são os próprios valores. Literais podem ser usados em qualquer parte de um programa Java.

Java utiliza o padrão UNICODE para armazenar as variáveis do tipo char. Nesse padrão, cada caracter ocupa 16 bits, podendo representar até 32.768 caracteres diferentes.

Os literais dessa categoria podem representar caracteres imprimíveis ou não. Abaixo segue uma relação de alguns deles :

'a-z'	– Uma letra de a - z	'\r'	– Retorno de carro
'A-Z'	– Uma letra de A- Z	'\b'	– Retrocesso do cursor
'\n'	– Nova linha	'\u006F'	– Valores Unicode
'\"'	– Aspas duplas	'\t'	– Tabulação
'\''	– Aspas simples		

- **Lógico**

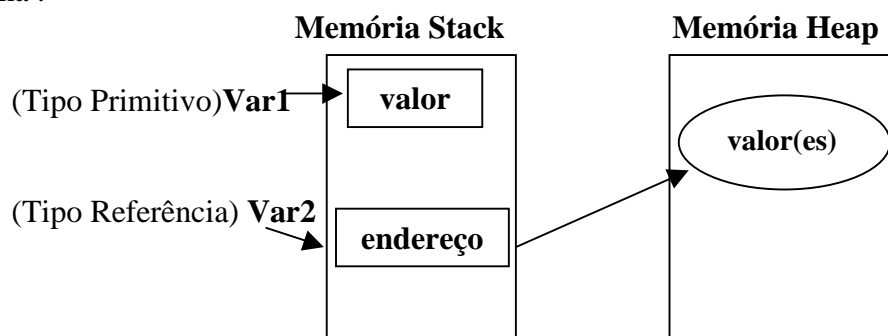
Essa categoria também possui apenas um tipo de dado que é o tipo **boolean**. Uma variável do tipo boolean pode armazenar apenas dois valores lógicos : **true** ou **false**. O Java não associa valores numéricos para true e false como ocorre em outras linguagens de programação.

Tipos de Dados Armazenados por Referência

Os tipos podem ser primitivos ou de referência. Os primitivos representam valores numéricos, caracteres individuais e valores booleanos. Os tipos de referência são representados por Classes, Interfaces e Arrays e podem ser definidos pelo usuário. A principal diferença entre os tipos primitivos e de referência é a memória que deve ser alocada, em tempo de execução, para uma variável de referência, pois o ambiente de execução não sabe quanta memória será necessária.

Na lista de tipos primitivos não encontramos um para tratar de strings de caracteres, isto porque em Java strings são manipuladas através de uma classe específica para isso, a classe **String**. Veremos mais sobre ela posteriormente.

Todo programa em Java possui dois tipos de memória : principal (stack) e auxiliar (heap). Uma variável do tipo primitivo armazena o seu conteúdo diretamente na memória principal. Já uma variável do tipo referência armazena o seu conteúdo na memória auxiliar e armazena na memória principal apenas o endereço onde foi armazenado o conteúdo da variável. Abaixo segue uma representação gráfica desse esquema :



3.3 – Escopo de Variáveis

O escopo de uma variável é o bloco de código dentro do qual ela é acessível e determina quando a variável é criada e destruída. O bloco é determinado por “{” e “}”.

De maneira geral, uma variável em Java pode ser um **atributo**, quando declarada na classe, ou uma variável **local** a um método, quando declarada dentro deste. Um atributo pode ser acessado por qualquer método de uma classe e só são destruídos quando todo o objeto for. Uma variável local pode ser acessada apenas dentro do método onde ela foi definida.

Mais especificamente, o escopo de uma variável está diretamente relacionada ao bloco onde ela está contida. Uma variável será visível apenas dentro do bloco imediatamente superior a ela. O exemplo abaixo ilustra essa regra :

```
void metodo ()
{  —————> Início do bloco de x
  int x;

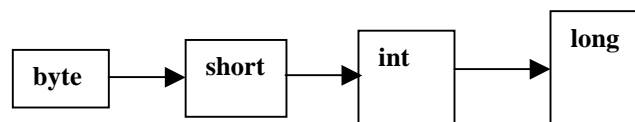
      {  ———> Início do bloco de y
        int y;
        x = 10; ———> OK ! A variável x está sendo acessada dentro de seu bloco !
      }  ———> Fim do bloco de y

  y = 20; ———> ERRO ! A variável y está sendo acessada fora de seu bloco !
}  —————> Fim do bloco de x
```

3.4 - Conversões Entre Tipos Primitivos de Dados

A conversão entre tipos primitivos de dados é uma prática bastante comum e necessária, porém, deve-se ter cuidado ao efetuá-la para não perder informações e obter um resultado não esperado.

Java automaticamente converte um valor de um tipo numérico para outro tipo de maior capacidade de armazenamento. Por exemplo, se tentarmos atribuir um byte para um short, o compilador automaticamente converte o byte em short antes de realizar a atribuição. O esquema abaixo simboliza as conversões automáticas :



O contrário não é verdadeiro. Para efetuarmos uma conversão de um tipo maior para outro de menor capacidade é necessário usar um mecanismo chamado **type cast**, caso contrário um erro será lançado. Isso é feito colocando-se o tipo pretendido para conversão entre parênteses em frente da expressão ou variável. Quando fazemos isso, forçamos o compilador a realizar a conversão. É importante estar ciente que isto pode resultar em perda de dados e em resultados imprevisíveis. Exemplo :

```
byte b1 = 127, b2 = 127, b3;
```

b3 = b1 + b2; —————> Erro, pois b1 + b2 resulta em 254

b3 = (byte) (b1 + b2); —————> Está correto, mas irá ocorrer perda de informação!

A tabela abaixo apresenta um resumo das possibilidades de cast, apontando os casos quando isso poderá causar perda de dados.

Tipo Original	Tipo Destino							
	byte	short	int	long	float	double	char	boolean
byte							C	X
Short	C, P						C	X
int	C, P	C, P					C, P	X
long	C, P	C, P	C, P		C, P	C	C, P	X
float	C, P	C, P	C, P	C, P			C, P	X
double	C, P	C, P	C, P	C, P	C, P		C, P	X
char	C, P	C	C	C	C	C		X
boolean	X	X	X	X	X	X	X	

C – Utilizar cast explícito P – Perda de magnitude ou precisão X – Java não permite conversão

3.5 – Operadores

Operadores são caracteres especiais usados para instruir o compilador a executar uma determinada operação em um operando.

- **Operador de Atribuição**

Operador	Nome	Exemplo
=	Atribuição	int var1 = 0, var2 = 0; var2 = var1 + 10; var1 = var2 = 50;

- **Operadores Aritméticos**

Operador	Nome	Exemplo	Resultado
+	Adição	23+29	52
-	Subtração	29-23	6
*	Multiplicação	0.5 * salario	
/	Divisão	100/50	2
%	Módulo (resto da divisão)	20%3	2

➤ Nas operações envolvendo byte, short ou char , estes valores são promovidos para int antes da operação, e o resultado será também um int. Exemplo :

byte b1 = 1, b2 = 2 , b3;

b3 = b1 + b2 —————> Erro pois o resultado é um int!

b1 = 100; b2 = 100;

b3 = b1 + b2; —————> Erro pois o resultado é um int!

b3 = (byte) (b1 + b2); —————> Com o cast não apresentará erro, porém ocorrerá perda de informação. Ao invés de ter 200 em b3, como ele é um byte o valor será -56 !

- Se a operação envolve um long em um dos argumentos do operador, então o argumento do outro lado é promovido para long e o resultado também será um long.

- **Operadores Relacionais**

Operador	Nome	Exemplo	Resultado
==	Igual	x == 10	
!=	Diferente	3 != 2	true
<	Menor	10 < 10	false
>	Maior	10 > 6	true
>=	Maior ou igual	3 >= 3	true
<=	Menor ou igual	7 <= 6	false

- **Operadores Lógicos**

Operador	Nome	Exemplo	Resultado
&&	AND	(0 < 2) && (10 > 5)	true
	OR	(10 > 11) (10 < 12)	true
!	NOT	!(1 == 4)	true
^	XOR	(1 != 0) ^ (2 < 3)	false

- **Atribuição Composta**

Operador	Nome	Exemplo (x = 10;)	Resultado
+=	mais igual	x += 10;	20
-=	menos igual	x -= 10;	10
*=	vezes igual	x *= 10;	100
%=	módulo igual	x %= 10;	0
/=	dividido igual	x /= 10;	0

- **Incremento e Decremento**

- Operador ++ incrementa de um. Ex : int var1 = 4; var1++; //var1 = 5
- Operador -- decrementa de um Ex : int var1 = 4; var1--; //var1 = 3
- Este decremento ou incremento pode ser feito antes ou depois da utilização da variável. Colocando-se antes, primeiramente será somado/subtraído 1 da variável e depois ela será utilizada. Ao contrário, se o operador vier após a variável, primeiramente ela será utilizada na expressão e o incremento ou decremento será feito depois disso. Ex :

```
int var1 = 2, var2;
var2 = var1++; //var2 vale 2
var2 = ++var1; //var2 vale 3
```

Precedência de Operadores

A **precedência** refere-se a ordem na qual os operadores são executados. Operadores de mesma precedência são executados de acordo com sua **associatividade**. Por exemplo, considere a seguinte linha de código :

```
int j = 3 * 10 % 7;
```

Os operadores * e % possuem a mesma precedência e associatividade da esquerda para direita. Assim, o * é executado primeiro, produzindo o resultado 2. Se fosse colocado um **parênteses** para modificar a precedência desta forma `int j = 3 * (10 % 7)`, o resultado final seria completamente diferente (9).

A tabela abaixo mostra os operadores Java em ordem de precedência, onde a linha 1 tem a mais alta precedência. Com exceção dos operadores unários, condicionais e o operador de atribuição, os quais são associativos da direita para esquerda, todos os outros operadores com a mesma precedência são executados da esquerda para direita.

Precedência	Operador	Comentários	Associatividade
1	++ -- !	Operadores Unários	R
2	* / %	Multi., Divi., Resto	L
3	+ -	Add, Sub.	L
4	<< >>	Shift	L
5	< > <= >=	Relacionais	L
6	= = !=	Igualdade	L
7	&	Bit/Logical AND	L
8	^	XOR (ou exclusivo)	L
9		Bit/Logical OR	L
10	&&	AND	L
11		OR	L
12	?:	Condicional	R
13	= op=	Atribuição	R

R = Direita L = Esquerda

3.6 – Constantes

Uma constante pode tanto ser definida como um atributo de classe como uma variável local. Uma constante uma vez que foi declarada e atribuído um valor para a mesma, não é permitido que outra atribuição seja efetuada, ocasionando um erro de compilação caso isso ocorra. A atribuição do valor inicial pode ser feita no momento da declaração, ou posteriormente em outra parte do programa.

Para declarar uma variável do tipo constante, devemos utilizar o modificador **final** dessa forma :

```
final <tipo> <identificador> [= valor];
```

Declare constantes com nomes descritivos, escritos com todas as letras em maiúsculo. Separe nomes internos com underscores (_). Exemplos :

```
final double MIN_WIDTH = 100.0;  
final long MAX_NUMBER_OF_TIMES;
```

Exercícios :

1 – Encontre os erros nas declarações de variáveis no código abaixo e os corrija.

1. byte sizeof = 200;
2. short \$mom = 43;
3. short hello mom;
4. double old = 0.1
5. double new = 78;
6. boolean consequence_2 = true;
7. char maine = "New England";

2 – Crie um programa seguindo as orientações abaixo. Faça tudo que pede dentro do método **main**.

- a) Declare duas variáveis para armazenar os valores de dois itens de venda. Os valores dos dois itens devem ser 2.95 e 3.50. Pense em nomes significativos para as variáveis e também considere qual deve ser seu tipo.
- b) Use o `System.out.println()` para mostrar o conteúdo de suas variáveis. Execute o programa e veja a saída. Apresente uma mensagem significativa como "O Item 1 custa 2.95 e ..." (Dica : utilize o operador + para concatenar o texto com o valor da variável).
- c) Declare uma outra variável para armazenar o custo total dos itens. Utilize o operador de adição para realizar o cálculo e imprima o resultado.
- d) Crie uma constante para armazenar a taxa de 8.25 % que deve ser cobrada sobre o valor total. Armazene o cálculo numa variável chamada `taxaCalculada` e imprima o resultado.
- e) Adicione ao valor de cada item o valor da taxa calculado. Use a menor linha de código possível para fazer isso. Some novamente os dois valores e atribua o resultado a uma variável chamada `novoCusto`.
- f) Crie uma variável booleana chamada `muitoCaro`. Escreva uma assertiva lógica para setar esta variável para `true` se `novoCusto` for maior que 10 e para `false` caso contrário. Imprima o valor obtido em `muitoCaro`. (Não utilize `if` para escrever a assertiva.)

3 – Declare uma variável chamada `Long` do tipo `long` e a inicialize com 100. Agora declare duas variáveis do tipo `int`, `int1` e `int 2`, e inicialize `int1` com 200. Agora faça `int2` receber `int1` mais `Long` e explique os resultados. Arrume o código para que não ocorra o erro anterior.

4 – Controle de Fluxo

A maioria dos programas tomam decisões que afetam seu fluxo. Os comandos que tomam essas decisões são chamados de comandos de controle.

4.1 – Comando if – else

Sintaxe :

```
if ( boolean_expr )
    comando1;
[else
    comando2;]
```

Expressão booleana formada pelos operadores lógicos e relacionais. Se a expressão booleana for verdadeira (true), executa comando1, caso seja falsa (false), executa comando2.

Exemplo :

```
if ( fim == true )
    System.out.println("Término!");
else
    System.out.println("Continuando...");
```

Para mais de um comando ser executado depois da declaração, utiliza-se o conceito de blocos, delimitados por { }.

Exemplo:

```
if ( fim == false ) && ( cont > 0 ){
    cont ++;
    System.out.println("Continuando...");
}
else {
    cont = 0;
    System.out.println("Término!");
}
```

É recomendado **sempre** usar blocos de comandos para facilitar a leitura e compreensão do código.

É possível também criar comandos if aninhados.

Exemplo :

```
if ( fim == false ) && ( cont < 0 ){
    cont++;
}
else if ( fim == false ) && ( cont > 0 ){
    cont--;
}
else {
    cont = 0;
    System.out.println("Término!!");
}
```

4.2 – Comando Switch

Utilizamos o comando switch para avaliar uma expressão contra vários resultados possíveis. Aceita **somente** um char, byte, short ou int como sua expressão de comparação. O valor é procurado nas declarações **case** que vem depois da declaração switch e o código adequado é executado.

Sintaxe :

```
switch ( expressão ) {
```

Deve resultar em char, byte, short ou int.

```
    case constant1 :  
        comando1;  
        break;
```

Deve ser um literal ou uma constante !

```
    case constant2 :  
        comando2;  
        break;
```

```
    ...
```

```
    [default :  
        comando_default;]
```

```
}
```

Quando encontra uma opção válida, os comandos são executados até um break ser encontrado, o qual transfere o controle para o comando seguinte ao comando switch. Ou seja, se um break não é especificado ao término de um case, todas as diretivas dos cases subsequentes serão executadas até que um break seja encontrado ou o comando switch termine.

Exemplos :

```
switch ( cmd ){  
  
    case 0: System.out.println("Item do menu 1");  
        menu = ++cmd;  
        break;  
    case 1: System.out.println("Item do menu 2");  
        menu = ++cmd;  
        break;  
    case 2: System.out.println("Item do menu 3");  
        menu = ++cmd;  
        break;  
    default: System.out.println("Comando invalido!");  
  
}
```

4.3 – Comando while

Implementa um loop para executar um bloco de comandos sucessivas vezes. A expressão de comparação é avaliada antes que o laço seja executado.

Sintaxe :

```
while ( boolean_expr ) {
```

```
    comandos;
```

```
}
```

Enquanto a expressão for verdadeira o laço será executado

Exemplo :

```
while ( i != 0 ){  
    salario = salario * 0.5;  
    i--;  
}
```

4.4 – Comando do - while

Utilizado quando se quer que o corpo do laço seja necessariamente executado pelo menos uma vez. A expressão de comparação é avaliada depois que o laço for executado.

Sintaxe:

```
do {  
    comandos;  
} while ( boolean_expr );
```

```
Exemplo:  
do {  
    salario = salario * 0.5;  
    i--;  
}while ( i != 0 );
```

4.5 – Comando for

Comumente, um loop for possui uma parte inicial com a inicialização das variáveis, seguida por uma expressão de comparação e depois a parte final com o incremento ou decremento das variáveis do laço.

Sintaxe :

```
for ( inicialização; condição; iteração ) {  
    comandos;  
}
```

```
Exemplo:  
for ( i = 0; i < 20; i ++){  
    salario = salario * 0.5;  
}
```

- Pode-se declarar variáveis na inicialização do for.

```
for (int i = 0 ; i < 10 ; i++)  
    comando;
```

- O único elemento realmente imprescindível é a condição. É possível construir laços for sem a inicialização e/ou o incremento.

```
int i = 2;  
for ( ; i<10 ; ){  
    comandos;  
    i++;  
}
```

- A inicialização e incremento podem ser feitos com mais de uma variável. Neste caso devem estar separados por vírgula.

```
for ( i = 0, j = 10 ; i < 8 && j > 2 ; i ++, j-- )  
    comando;
```

4.6 – Comando break

O break é utilizado para transferir o controle para o final de uma construção de laço (for, do-while, while) ou de um switch. O laço vai encerrar independentemente de seu valor de comparação e o comando após ele será executado. Exemplo :

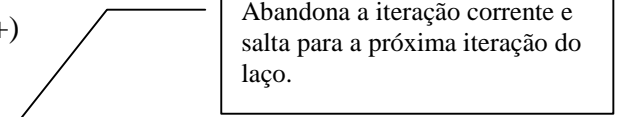

```
int i = 0;
while (true) {
    System.out.println(i);
    i++;
    if ( i > 10 ) break;
}
... ←
```

4.7 – Comando continue

Quando o comando continue é encontrado (somente dentro de um laço), ele pára a execução naquele ponto e a transfere de volta para o início do laço para dar início a uma nova iteração.

Exemplo :

```
for (int i = -10; i<10; i++)
{
    if ( i == 0 )
        continue;
    System.out.println(1/i); //Não deve ser executado o cálculo 1/0 !!
}
```



Abandona a iteração corrente e salta para a próxima iteração do laço.

Exercícios :

1 – Identifique e corrija os erros dos códigos abaixo :

```
int x = 3, y = 5; 1
if (x >= 0)
    if (y < 0)
        System.out.println("y é menor que x");
else
    System.out.println("x é negativo");
```

```
int x = 7; 2
if (x = 0)
    System.out.println("x é zero");
```

```
int x = 15, y = 24; 3
if ( x %2 == 0 && y%2 == 0 ) ;
    System.out.println("x e y são pares");
```

2 – Determine o número de dias num mês. Você deve usar um comando switch para setar um inteiro com o número de dias no mês especificado. Por enquanto adicione todo o código no método main().

- a) Declare três inteiros, um para manipular os dias (1-31), outro os meses (1-12) e um terceiro para manipular o ano. Inicialize estas variáveis com um dia da sua escolha.
- b) Imprima a data no formato dd/mm/yyyy.
- c) Declare uma variável para armazenar o número de dias que contém o mês escolhido. Então, usando o comando switch, determine o valor para esta variável. São 30 dias em Abril, Setembro, Junho e Novembro e 31 nos restantes exceto para Fevereiro que tem 28 (ignore o ano bissexto).

Experimente executar o programa com diferentes valores para o mês. O que acontece se ele for executado com um mês inválido como 13 ? Para 27 de Janeiro de 2000 a saída deve ser algo como

```
27/1/2000
31 dias no mês
```

3 – Use um laço for para mostrar datas.

- a) Usando um laço for, estenda seu programa de forma que ele imprima todas as datas entre a que você especificou até o final daquele mês. Para 27/1/2000 a saída deve ser

```
27/1/2000
28/1/2000
29/1/2000
30/1/2000
31/1/2000
```

- b) Teste o programa para verificar se ele está funcionando corretamente com várias datas diferentes.
- c) Agora o modifique para que ele imprima no máximo 10 datas. Por exemplo, se a data for 19/1/2000 o programa deve imprimir apenas as datas 19/1/2000 até 28/1/2000. Já se a data for 30/1/2000, deverá ser impresso apenas 30/1/2000 e 31/1/2000.
- d) Compile seu programa, corrija os erros se ocorrer algum, e teste com várias datas para ter certeza que está funcionando corretamente.

4 – Determine se o ano especificado é um ano bissexto.

- a) Crie uma expressão booleana para testar se o ano é bissexto. Um ano é bissexto se :

ele é divisível por 4, e
ele não é divisível por 100, ou é divisível por 400

- b) Modifique o comando switch da questão 2 para tratar de anos bissexto. Lembre-se que o ano Fevereiro tem 29 dias em anos bissextos e 28 nos outros.
- c) Compile e teste seu programa. A tabela abaixo apresenta alguns anos que são e outros que não são bissextos para você poder testar seu programa :

Anos Bissextos	Anos não Bissextos
1996	1997
1984	2001
2000	1900
1964	1967

5 – Considere que a data que você escolheu é a data de empréstimo de uma mercadoria e que é necessário saber qual a data de devolução da mesma. A data de devolução é a data atual mais 3 dias.

- a) Declare três variáveis para controlarem o dia , mês e ano da data de devolução e uma outra com a quantidade de dias de empréstimo (inicializada com 3).
- b) Crie o código para calcular a adição do tempo de empréstimo com a data escolhida. A saída deve se assemelhar a algo como :

Data do Empréstimo : 27/2/2001
Número de Dias : 3
Data da Devolução : 2/3/2001

Preste atenção na saída acima. A soma de três dias na data de empréstimo resultou na passagem de um mês para outro. Além disso, como o ano 2001 não é bissexto e o mês de empréstimo foi fevereiro, ao somar-se 3 dias os dias que se passaram foram : 28/2 – 1/2 – 2/3 . Você deve estar atento também para :

- Se o empréstimo passar de um mês para outro o mês deve ser incrementado
- Se o empréstimo passar de um ano para outro o ano deve ser incrementado

5 – Métodos

Quando definimos um objeto num programa orientado a objetos, implementamos todo o comportamento desse objeto em um ou mais métodos. Um método em Java é equivalente a uma função, subrotina ou procedimento em outras linguagens de programação.

Não existe em Java o conceito de métodos globais. Todos os métodos devem **sempre** ser definidos dentro de uma classe.

A sintaxe para construção de um método é a seguinte :

```
[modificador] tipo_retorno identificador ([argumentos]) {  
  
    //Corpo do método  
  
}
```

5.1 – Modificadores de Acesso

Java controla o acesso a **atributos** e **métodos** através do uso dos modificadores de acesso. São eles :

- **public** : É o menos restritivo de todos. Atributos e métodos declarados como public em uma classe podem ser acessados pelos métodos da própria classe, por classes derivadas desta e por qualquer outra classe em qualquer outro pacote (veremos mais sobre pacotes e classes derivadas posteriormente).
- **protected** : Atributos e métodos definidos como protected são acessíveis pelos métodos da própria classe e pelas classes derivadas.
- **private** : É o mais restritivo. Atributos e métodos declarados como private só podem ser acessados pelos métodos da própria classe.

Quando nenhum modificador é definido (acesso do tipo “package”), os atributos e métodos podem ser acessados pelos métodos da própria classe, pelas classes derivadas e por qualquer outra classe dentro do mesmo pacote.

É importante destacar que esse controle não se aplica às variáveis locais aos métodos.

5.2 – Nome de Métodos

O nome de um método deve começar com uma letra (a-z/A-Z), um underscore (_), ou um sinal de dólar (\$). Os caracteres subsequentes podem incluir os dígitos de 0 a 9.

- **Convenção** : Use verbos para nome de métodos. Faça a primeira letra do nome minúscula com cada letra inicial interna maiúscula. Por exemplo : getUsername(), getMaxPrice().

5.3 – Argumentos

- Um método pode ter zero ou mais argumentos (ou parâmetros).

- Caso não tenha argumentos, é necessário informar os parênteses vazios tanto na definição como na chamada do método.
- O nome do método acrescido de seus parâmetros é denominado **assinatura do método**.
- Cada argumento deve ser declarado como define-se uma variável, especificando o seu tipo e nome.
- Caso seja mais de um argumento, cada declaração deve estar separada por vírgula.

Exemplos :

```
public void setAge(int newAge)
{
    age = newAge;
}
```

```
public void displayAge()
{
    System.out.println("Age is : " + age);
}
```

```
public void setUser(int newAge, String newName)
{
    age = newAge;
    name = newName;
}
```

5.4 – Retornando Valor a Partir de um Método

É possível associar um valor de retorno a um método. Para isso, é preciso definir, na frente do nome do método, o tipo do valor a ser retornado. O tipo pode ser um primitivo ou então uma classe ou interface. Caso o método não retorne valor algum, é obrigatória a utilização do tipo **void** na assinatura do método.

Exemplos :

```
public int getAge () {
    return age;
}
```

```
public String getName () {
    return name;
}
```

O comando **return** é utilizado para passar o valor requerido de volta para quem chamou o método e é **obrigatório** quando o tipo de retorno não é void. Ele deve receber um único valor ou uma expressão que deve ser compatível com o tipo de retorno especificado para o método. Quando o comando é encontrado, o método termina sua execução imediatamente, ignorando todas as instruções subsequentes. Um comando return sem valor de retorno retorna o tipo void, mas não é obrigatória sua utilização nestes casos.

5.5 – Passando Parâmetros para um Método

- ❖ Passando um Primitivo : Quando um valor primitivo é passado na chamada de um método, uma **cópia** deste valor é criada e atribuída para o argumento do método responsável por recebê-la. Se o método mudar este valor, apenas o valor do argumento local ao método é afetado. Quando o método terminar sua execução, o

valor original da variável utilizada para passar o valor primitivo na chamada do método permanecerá inalterado.

```
int num = 10;
```

```
incrementa (num); —————> Imprimirá 11
```

```
public void incrementa (int num) {  
    num++;  
    System.out.println("num : " + num);  
}
```

```
System.out.println("num : " + num); —————> Imprimirá 10
```

- ❖ **Passando a Referência de um Objeto :** Quando o tipo passado para o método não for um primitivo mas sim um objeto, esse comportamento muda. Quando passamos um objeto, uma referência ao objeto original é passada ao invés de uma cópia do objeto. A referência contém o endereço de memória onde está contido o objeto original e qualquer modificação feita pelo método no argumento que recebeu esta referência afetará também o objeto original.

```
StringBuffer texto = new StringBuffer("Texto inicial!");
```

```
System.out.println("O valor de texto é : " + texto);
```

```
concatMsg(texto);
```

```
System.out.println("O valor de texto é : " + texto);
```

Texto inicial!

Texto inicial! Esta frase foi acrescentada ao objeto dentro do método!

StringBuffer

```
public void concatMsg (StringBuffer msg){  
    msg.append(" Esta frase foi acrescentada ao objeto dentro do método!");  
}
```

Exercícios :

1) Digite e execute o código abaixo, onde um objeto da classe String é passado como parâmetro para um método. Ocorreu algo de inesperado? Procure explicar o que.

```
public class testeMetodos{

    public testeMetodos() {
    }

    public void concatMsg (String msg){

        System.out.println("O valor de texto é : " + msg.concat("Esta frase foi
            acrescentada ao objeto dentro do método!"));
    }

    public static void main(String[] args) {

        testeMetodos testeMetodos1 = new testeMetodos ();

        String texto = new String("Texto inicial!");
        System.out.println("O valor de texto é : " + texto);
        testeMetodos1.concatMsg(texto);
        System.out.println("O valor de texto é : " + texto);

    }
}
```

6 – Classes e Objetos em Java

A orientação a objeto é a técnica de desenvolvimento e modelagem de sistemas que facilita a construção de programas complexos a partir de conceitos que nos permitem modelar o mundo real tão próximo quanto possível da visão que temos desse mundo. Enquanto o desenvolvimento estruturado baseia suas abstrações para compreensão de um problema nos procedimentos necessários para sua solução, a orientação a objetos nos permite modelá-lo num nível maior de abstração, através da identificação natural que fazemos das entidades que o compõe, caracterizando-as a partir de certas propriedades que definem seus principais aspectos estruturais e comportamentais.

“**Objetos** são um conjunto de dados mais um conjunto de procedimentos que representam a estrutura e o comportamento inerentes às entidades, concretas ou abstratas, do mundo real.”

Entidades concretas são aquelas que representam objetos concretos tais como um carro ou um avião. Possuem uma coleção de atributos (material, características mecânicas, custos, métodos construtivos) e uma coleção de métodos próprios que manipulam e fornecem respostas a partir dos valores desses atributos (reações, deslocamentos, custo total). Além destas, entidades não tão concretas, como a representação de elementos geométricas num plano, por exemplo, são tratadas como objetos. Nesse caso, a estrutura é composta por atributos que constituem variáveis como o número de lados e valores dos ângulos internos e o comportamento é ditado por métodos que informam a área ou algum outro valor que servirá de contribuição para a solução de um problema matemático. Na Programação Orientada a Objeto (POO), nada impede que a essência de um objeto seja o processamento ao invés da estruturação ou vice-versa. Na verdade, objetos podem ser, em princípio, o modelo de qualquer coisa, desde um processamento puro até um dado puro, sendo todos, sob a ótica da POO, igualmente considerados objetos.

O paradigma da orientação a objetos prevê a definição de objetos como principal artifício na modelagem de uma aplicação. Sendo assim, a primeira atividade a ser executada na implementação de uma aplicação orientada a objetos é a identificação dos objetos que a compõe e suas propriedades. Essa identificação sempre depende do propósito, porque é o propósito que permite determinar quais informações serão representadas e que operações serão executadas pelos objetos. Em outras palavras, a identificação de quais atributos e métodos terá um objeto depende sempre do contexto da aplicação, de como pretendemos usar o objeto. Só tendo isso em mente podemos levantar os atributos e métodos relevantes ao objeto e suprimir aqueles irrelevantes para o sistema em questão.

A **classe** é a construção de linguagem mais comumente utilizada para implementar um objeto em linguagens de programação orientada a objeto. Objetos com a mesma estrutura e o mesmo comportamento pertencem a mesma classe de objetos. O conjunto de dados de um objeto chamamos **atributos** e o conjunto de procedimentos de **métodos** do objeto. Uma classe, portanto, é uma descrição dos atributos e dos métodos de determinado tipo de objeto. **Em termos de programação, definir classes significa formalizar um novo tipo de dado e todas as operações associadas a esse tipo, enquanto declarar objetos significa criar variáveis do tipo definido.**

Uma **instância** é um objeto gerado por uma classe. A classe é apenas a representação de um objeto enquanto a instância de uma classe é o objeto propriamente dito, com tempo

e espaço de existência. O conteúdo de um objeto, ou seu **estado interno**, é definido por suas **variáveis de instância** (atributos), sendo que dois objetos diferentes da mesma classe podem possuir valores diferentes para seus atributos e conseqüentemente apresentarem estados internos diferentes.

Exemplo:

```
public class Ponto
{
    double x;
    double y;
```

```
void girar (int grau, Ponto ponto) {
```

```
    //Código para girar esse ponto em torno de outro
```

```
}
```

```
Ponto P1;
```

Os atributos ou variáveis de instância formam o **estado interno** de um objeto. Eles definem os dados que um objeto deve possuir.

Os métodos representam as **operações** que um objeto pode executar. Estas operações podem, dentre outras coisas, alterar o estado interno do objeto.

Aqui, estamos definindo uma variável do **tipo** Ponto. Dizemos que P1 é uma **instância** da classe Ponto, isto é, corresponde a um objeto desta classe.

Estudo de Caso

Vamos analisar o problema abaixo extraído da computação gráfica:

“Desenhar um círculo na tela do computador. Deve ser possível configurar a cor que o círculo estará preenchido bem como seu diâmetro.”

Dado o problema e seus requisitos, devemos proceder com a modelagem do mesmo, isto é, analisar suas características para decidir qual a melhor forma de representar a solução numa abordagem orientada a objetos. Vamos supor que não esteja disponível uma rotina que execute o que se pede. A única coisa que sabemos é como imprimir na tela do computador uma linha dado seus pontos de origem e destino. Sendo assim, baseado no que sabemos fazer, teremos que analisar o problema até conseguirmos decompô-lo em estruturas mais primitivas que, quando combinadas, irão produzir uma solução. Os passos de raciocínio poderiam ser estes:

- Deve ser possível configurar o diâmetro do círculo e o diâmetro mede duas vezes o raio.
- O raio pode ser representado por uma linha cujo ponto inicial é o centro do círculo e o ponto final sua extremidade.
- Uma linha é formada por dois pontos, cada um com suas coordenadas (x,y) e possui uma cor.
- Uma cor é definida por seus componentes RGB (Red, Green, Blue).
- Bem, se sabemos pintar uma linha na tela, podemos definir o raio do círculo e ir pintando uma linha do lado da outra, mantendo as coordenadas do centro constantes e calculando as coordenadas da extremidade de forma que represente um grau de distância da linha anterior até completar 360°.

De acordo com a descrição acima, podemos identificar 4 entidades com características próprias e que são candidatas a tornarem-se objetos da nossa aplicação:



Em Java, os dados e os métodos pertinentes a cada uma dessas entidades poderiam ser modelados nas seguintes classes:

```
public class Cor
{
    double Red;
    double Green;
    double Blue;
}

public class Ponto
{
    double x;
    double y;
    void girar (int grau, Ponto p) {

        //Código para girar o ponto em torno de outro ponto p a quantidade definida em
        //grau
    }
}
```

Componentes RGB da cor

Coordenadas do ponto

Método que implementa o comportamento do ponto de “saber girar em torno de outro”.

```
public class Linha
{
    Ponto a;
    Ponto b;
    Cor color;
    void imprimeLinha() {

        //Código para imprimir essa linha na tela
    }
}
```

Uma classe pode (deve) ser construída a partir de outras. A este relacionamento dá-se o nome de **agregação**. Neste caso, a classe Linha utiliza as classes Ponto e Cor para compor seus atributos. Ela também possui um método que implementa a operação de impressão da linha.

```
public class Círculo
{
    final int GRAU = 360;
    Linha raio;
    Cor cor;

    void criaCírculo () {
```

A classe Círculo é a abstração de mais alto nível desse problema. Na verdade, criamos todas as outras classes para conseguirmos no final gerar um círculo na tela. O fato de quebrarmos um grande objeto (problema) em um número de objetos de mais baixo nível, consiste numa abordagem eficaz para compreendermos problemas complexos e possibilita freqüentemente a reutilização destes objetos na solução de outros problemas.

```
    raio.imprimeLinha();
    for (int i = 0 ; i < GRAU ; i++)
    {
        //gira o ponto b do raio (extremidade) i graus em relação ao ponto a (centro)
        raio.b.girar(i, raio.a);
        //imprime a linha resultante
        raio.imprimeLinha();

    } //fim for
} //fim criaCirculo
} //fim Círculo
```

Este método utiliza a Linha que representa o raio do círculo e gera um círculo da mesma cor da linha. A funcionalidade do método é basicamente executada por chamadas a métodos de outras classes, simplificando sua implementação.

A classe `Círculo` foi composta basicamente pela agregação de outras classes que implementam funcionalidades mais específicas. Essa abordagem *bottom-up* no tratamento de um problema, onde pequenos módulos são agrupados para gerarem outros maiores e com funcionalidades mais complexas, facilita a implementação, aumenta a legibilidade do código e melhora em muito a manutenibilidade da aplicação.

6.1 – Encapsulamento

Diferente da abordagem estruturada, onde dados e procedimentos são definidos de forma separada na código, na programação orientada a objeto os dados e procedimentos que manipulam estes dados são definidos numa unidade única, o objeto. Isso possibilita uma melhor modularidade do código, porém, a idéia principal é poder utilizar os objetos sem ter que se conhecer sua implementação interna, que deve ficar escondida do usuário do objeto que irá interagir com este apenas através de sua **interface**.

“À propriedade de se implementar dados e procedimentos correlacionados em uma mesma entidade e de se proteger sua estrutura interna escondendo-a de observadores externos dá-se o nome de **encapsulamento**. “

O objetivo do encapsulamento é separar o usuário do objeto do programador do objeto e seus principais benefícios são:

- possibilidade de alterar a implementação de um método ou a estrutura de dados escondidos de um objeto sem afetar as aplicações que dele se utilizam;
- criação de programas mais modulares e organizados, o que possibilita um melhor reaproveitamento do código e melhor manutenibilidade da aplicação.

Via de regra, as variáveis de instância declaradas em uma definição de classe, bem como os métodos que executam operações internas sobre estas variáveis, se houverem, devem ser escondidos na definição da classe. Isso é feito geralmente através de construções nas linguagens de programação conhecidas como modificadores de acesso, como por exemplo o **public**, **protected** e **private** do Java. Quando definimos uma classe, é recomendado (para alguns é uma regra sagrada) que declaremos públicos apenas os métodos da sua interface. É na **interface** (ou protocolo) da classe que definimos quais mensagens podemos enviar às instâncias de uma classe, ou seja, quais são as operações que podemos solicitar que os objetos realizem. Por exemplo, na classe `Ponto` acima, deveríamos ter feito seus atributos protegidos e apenas o método público:

```
public class Ponto
{
    private double x;
    private double y;

    public void girar (int grau, Ponto p) {
        //Código para girar o ponto em
        //torno de outro ponto p a quantidade
        //definida em grau    }
    public void setX (double k) { x = k; }
    public void setY (double k) { y = k; }
    public double getX () { return x; }
    public double getY () { return y; }
}
```

Os métodos públicos
constituem a interface da
classe.

É comum em programação orientada
a objeto definir métodos gets e sets
que provêm acesso aos dados
protegidos da classe.

6.2 – Criando Objetos e Acessando Dados Encapsulados

Criamos objetos em Java de forma muito similar a criação de variáveis de tipos primitivos. Se uma aplicação quisesse usar a classe `Círculo`, poderia declarar uma variável deste tipo da seguinte forma:

```
Círculo círculo; → Como Java é case-sensitive pode-se declarar desta forma.
```

Esta sentença cria uma variável `círculo` do tipo `Círculo`.


Entretanto, isso não é suficiente para acessarmos os métodos e atributos públicos da classe. A sentença acima somente declara uma variável mas não cria um objeto da classe especificada. Em Java, objetos são criados usando o operador **new** da seguinte forma:

```
Círculo círculo;  
círculo = new Círculo();
```

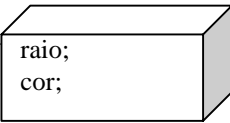
O operador **new** cria uma instância da classe e retorna a referência do novo objeto. Como vimos, todos os objetos em Java são tratados através da referência ao endereço de memória onde o objeto está armazenado. O operador **new** realiza três tarefas:

1. Aloca memória para o novo objeto
2. Chama um método especial de inicialização da classe chamado **construtor**
3. Retorna a referência para o novo objeto

É importante compreender o que ocorre na declaração da variável e na inicialização da variável. Quando fazemos

```
Círculo círculo; →  O null é um valor válido em Java e permite comparações como:  
if (círculo == null)
```

é reservado uma porção da memória principal do Java (stack) para armazenar o endereço na memória auxiliar (heap) onde o objeto será armazenado. Como apenas com a declaração da variável o objeto ainda não existe, o conteúdo inicial dela será o valor nulo (**null**), indicando que ela ainda não se refere a nenhum objeto. Apenas após a inicialização

```
círculo = new Círculo(); 
```

é que uma variável de um tipo não primitivo estará valendo algo e através dela será possível acessar os dados e operações do objeto em questão.

Uma vez um objeto tendo sido criado, seus métodos e atributos públicos podem ser acessados utilizando o identificador do objeto (variável que armazena sua referência) através do operador **ponto**:

```
<identificador>.<atributo>  
<identificador>.<método>
```

A aplicação que criou o objeto `Círculo` acima pode solicitar ao objeto que ele se desenhe fazendo :

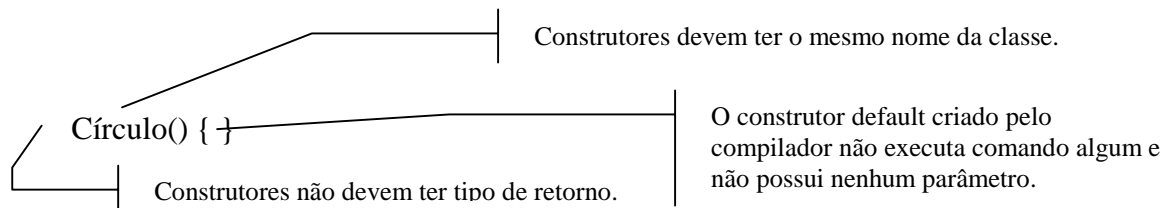
```
círculo.criaCírculo();
```

6.3 – Construtores

O construtor de uma classe é um método especial com as seguintes características:

- É o primeiro método executado por um objeto
- Deve ter o mesmo nome da classe
- Não pode ter um tipo de retorno
- É utilizado quando um objeto é instanciado através do operador **new**
- Na maioria das vezes é declarado como **public**

É obrigatório uma classe conter ao menos um construtor. Se uma classe for definida sem a presença de um, o compilador Java irá criar automaticamente um construtor default da seguinte forma:



Na classe `Círculo` não definimos nenhum método construtor, mas mesmo assim podemos criar a variável `círculo` chamando o construtor default da classe.

Como os construtores são os primeiros métodos executados por um objeto eles geralmente são utilizados para inicializar variáveis de instância da classe, porém, nada impede que executem qualquer operação que se queira.

Vamos alterar nossas classes acima e definir construtores para elas:

```
public class Cor
{
    ▶ Cor(double vermelho, double verde, double azul) {
        Red = vermelho;
        Green = verde;
        Blue = azul;
    }
    double Red;
    double Green;
    double Blue;
}
```

```
public class Linha
{
    Ponto a;
    Ponto b;
    Cor color;
    void imprimeLinha() { ... }
    ▶ Linha(Ponto ori, Ponto dest, Cor c)
    {
        a = ori; b = dest; color = c;
    }
}
```

```
public class Ponto
{
    private double x;
    private double y;
    ▶ Ponto(double p1, double p2) { x = p1; y = p2; }
    public void girar (int grau, Ponto p) {
        //Código para girar o ponto em
        //torno de outro ponto p a quantidade
        //definida em grau    }
    public void setX (double k) { x = k; }
    public void setY (double k) { y = k; }
    public double getX () { return x; }
    public double getY () { return y; }
}
```

```
public class Círculo
{
    final int GRAU = 360;
    Linha raio;
    Cor cor;
```

Quando definimos algum construtor para a classe, temos que implementar manualmente o construtor default caso queiramos usá-lo.

```
▶ Círculo() {}
```

```
▶ Círculo(Ponto centro, Ponto extremidade, Cor c) {
    raio = new Linha(centro, extremidade, c);
    cor = c;
}
```

Uma classe pode ter mais de um construtor.

```
▶ Círculo (double x1, double y1, double x2, double y2, Cor c)
{
    raio = new Linha(new Ponto(x1, y1), new Ponto(x2, y2), c);
    cor = c ;
}
```

```
void criaCírculo () {
```

```
    raio.imprimeLinha();
    for (int i = 0 ; i < GRAU ; i++)
    {
        //gira o ponto b do raio (extremidade) i graus em relação ao ponto a (centro)
        raio.b.girar(i, raio.a);
        //imprime a linha resultante
        raio.imprimeLinha();
```

```
    }//fim for
```

```
}//fim criaCirculo
```

```
//fim Círculo
```

Dado os construtores definidos, vamos analisar as possibilidades de implementação para se construir um círculo:

- Usando o construtor Círculo(): Se quisermos criar um círculo usando o construtor default, teremos que criar antes uma linha e uma cor e então atribuímos seus valores manualmente para o círculo.

```
Círculo círculo = new Círculo();
Cor c = new Cor (1.0, 2.0, 1.0);
Ponto p1 = new Ponto(0,0);
Ponto p2 = new Ponto(2,0);
Linha l = new Linha (p1, p2, c);
círculo.raio = l;
círculo.cor = c;
círculo.criaCírculo();
```

- Usando o construtor Círculo(Ponto, Ponto, Cor):

```
Cor c = new Cor (1.0, 2.0, 1.0);
Ponto p1 = new Ponto(0,0);
Ponto p2 = new Ponto(2,0);
Círculo círculo = new Círculo(p1,p2,c)
círculo.criaCírculo();
```

- Usando o construtor `Círculo(double x1, double y1, double x2, double y2, Cor c)`:

```
Cor c = new Cor (1.0, 2.0, 1.0);
Círculo círculo = new Círculo(0, 0, 2, 0, c);
círculo.criaCírculo();
```

Definir mais de um construtor para a mesma classe fornece maior flexibilidade para sua utilização.

6.4 – O Ponteiro *this*

Todos os métodos de instância recebem um argumento implícito chamado *this*, que pode ser usado dentro do método para se referir ao objeto ao qual aquele método pertence. Mas para que iríamos querer um ponteiro para o próprio objeto se dentro de um método é possível acessar qualquer outro método ou atributo da classe?

Existem duas situações onde devemos usar uma referência explícita ao *this*:

- Quando o nome de uma variável de instância é escondido por um argumento de um método. Por exemplo, se tivéssemos criado o construtor

`Círculo(Linha r)`

como

```
Círculo (Linha raio) {
    raio = raio;
    cor = raio.color ;
}
```

a atribuição `raio = raio` não ia causar o efeito desejado pois onde `raio` aparece dentro do método está se referindo ao argumento e não a variável de instância. Para este código funcionar apropriadamente teríamos que usar o ponteiro *this* da seguinte forma:

```
Círculo (Linha raio) {
    this.raio = raio;
    cor = raio.color ;
}
```

- Quando é preciso passar uma referência ao objeto corrente como argumento a um outro método. Vamos supor a existência de uma classe chamada `Transforma` capaz de realizar transformações geométricas num círculo qualquer. Vamos supor também que a classe `Círculo` possua um método chamado `Duplica()` para duplicar seu tamanho e que este método utiliza um objeto da classe `Transforma`. O código de `Duplica()` é:

```
Duplica() {
    Transforma t = new Transforma();
    t.duplicar(this);
}
```

| Passa o próprio objeto como parâmetro.

Exercícios :

- 1) Para iniciar a prática deste capítulo, você deve modelar o problema descrito abaixo, que descreve o negócio da ACME Vídeo Locadora. Serão apresentadas inúmeras informações que ajudarão você a determinar as classes, métodos, atributos e associações que compõem o negócio. Fique livre para escolher o formato que pretende usar para comunicar o que você encontrou.

A ACME vídeo locadora está no negócio de aluguel de vídeos e estamos interessados neste momento em automatizar o processo de retirada das fitas. Este processo consiste do cliente, após ter escolhido as fitas que deseja, levá-las até o balcão de retirada. O funcionário responsável irá solicitar o número do cartão de associado do cliente e depois cadastrar os IDs dos itens que este escolheu.

Nossos clientes devem se cadastrar como associados antes de poderem efetuar qualquer locação. Nós armazenamos informações como nome, endereço e telefone e cada cliente possui um ID único que utilizará para efetuar as locações.

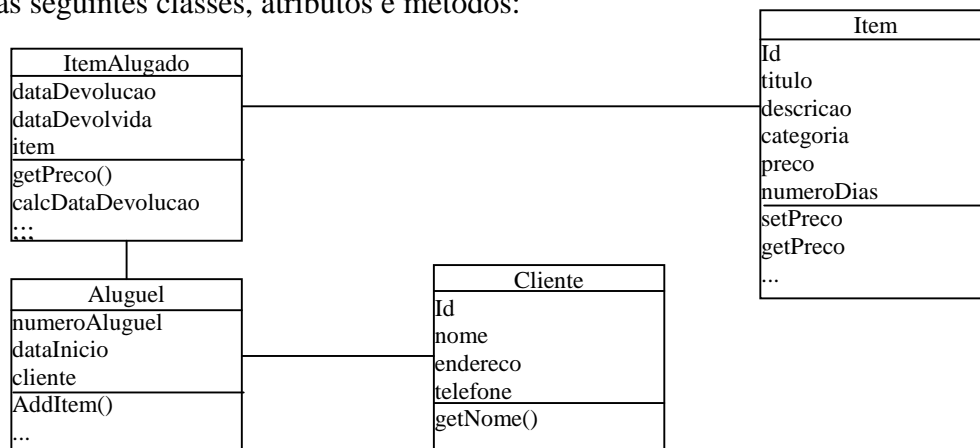
Quando uma locação é efetivada, damos um número único para ela que será utilizado posteriormente para mantermos o controle da mesma. A efetivação de uma locação possui informações como: data de início, o cliente que solicitou e uma lista dos itens alugados.

Para cada item alugado controlamos a data que deve ser devolvido e a data que realmente foi devolvido. A data de devolução é dependente de quantos dias um item pode permanecer emprestado, sendo assim, cada item pode ter uma data diferente.

Os itens para empréstimo são atualmente apenas fitas de vídeo. Mas pretendemos alugar em breve jogos e DVDs. Cada item para empréstimo possui as seguintes informações: preço do aluguel, o número de dias que pode ser emprestado, o título do filme, uma breve descrição (sinopse) e uma categoria (ação, aventura, etc).

- a) Identifique as classes que compõem o negócio da ACME Locadora. Escreva uma breve descrição sobre ela para ter certeza de sua relevância para o negócio.
- b) Identifique os atributos de cada uma das novas classes. Lembre-se que um atributo pode ser outra classe.
- c) Defina alguns comportamentos para cada uma das classes.

- 2) A partir da análise do exercício anterior, provavelmente (:-)) você vai ter chegado nas seguintes classes, atributos e métodos:



Agora vamos iniciar a implementação da aplicação ACME Vídeo Locadora. Você irá criar um projeto e adicionar as classes identificadas. Estas classes formam a base para a aplicação que será desenvolvida no restante deste curso.

- a) Abra o projeto chamado ACMEVideo que deve conter um método main.
- b) Crie a classe Cliente de forma que ela não contenha um método main(), seja pública e contenha um construtor default. Adicione então os atributos privados da classe:

```
int id
String nome
String endereco
String telefone
```

Adicione também métodos públicos para setar e pegar cada um dos atributos privados. Utilize um padrão de nome como setId, setNome, getId, getNome, etc:

```
void setId(int id){ this.id = id;}
```

No método main() do projeto crie dois clientes e estipule valores para seus atributos utilizando os métodos públicos da classe.

Até este momento não há uma forma de apresentar o conteúdo de um objeto cliente. Para remediar esta deficiência adicione à classe Cliente o método toString(), que retorna uma String contendo o id, nome do cliente, endereço e telefone. Você pode usar o operador + para concatenar Strings. Chame este método e imprima a String de retorno usando System.out.println(...).

```
public String toString(){
    return (atributo1 + “ ” + atributo2 ...);
}
```

- c) Crie a classe Item como pública, sem construtor e sem método main(). Defina seus atributos privados e métodos para setar (setId) e ler (getId) os atributos.

```
int getId(){ return id;}
```

Os atributos desta classe são:

```
int id
String titulo
String descricao
String categoria
double preco
int numeroDias
```

Crie no método main do projeto ACMEVideo alguns Itens para testar os métodos.

Adicione à classe um método toString() que apresenta as variáveis título, descrição e categoria.

- d) Vamos agora adicionar um outro construtor na classe Cliente. Este deve inicializar os atributos nome, endereço e telefone do objeto.

Modifique o método main() da aplicação ACMEVideo para usar o novo construtor. Compile e teste o programa imprimindo o conteúdo dos objetos Cliente criados para ter certeza que o construtor os inicializou corretamente.

- e) Adicione um construtor default que não faz nada para a classe Item e também um que inicialize os atributos título, descrição, categoria, preço e numeroDias.

Modifique novamente o método main() para usar este novo construtor e teste a aplicação como feito com a classe Cliente.

7 – Mais Sobre Classes e Objetos

7.1– Pacotes

Um pacote é uma coleção de classes que estão logicamente relacionadas. Um pacote consiste de todas as classes Java contidas num diretório. Por exemplo, as classes da aplicação ACME Vídeo que estamos desenvolvendo em nossos exercícios localizam-se no diretório do projeto chamado ACMEVideo e conseqüentemente fazem parte do pacote ACMEVideo. O nome do pacote deve ter o mesmo nome do diretório onde as classes estão armazenadas e a declaração do pacote a qual uma classe pertence deverá ser feita no início do arquivo fonte da classe da seguinte forma:

```
package ACMEVideo;
```

Como sabemos, Java tem uma vasta biblioteca² de classes que podem ser utilizadas para ajudar-nos no desenvolvimento de nossos programas. Para termos acesso a uma biblioteca Java em nossa aplicação devemos utilizar o comando **import** da seguinte forma:

```
import java.util.Vector;
```

se quisermos utilizar apenas a classe Vector do pacote util, ou

```
import java.util.*;
```

se quisermos deixar todas as classes do pacote java.util disponíveis para nossa aplicação. O comando **import** deve ser especificado antes da definição de todas as classes. O caminho java.util corresponde ao diretório onde as classes do pacote util foram armazenadas, colocando-se o ponto (.) para referenciar a passagem de um diretório para outro.

É importante ter em mente que, diferentemente do C, acrescentar pacotes num programa Java não implica em as classes contidas neles serem carregadas dentro do programa. Isto apenas instrui ao compilador e à JVM onde procurar pelas classes que serão necessárias para a execução. Resumindo, inserir comandos **import** não degrada a performance da aplicação.

Além de possibilitar uma melhor organização, os pacotes servem para o Java desempenhar dois controles importantes:

1. Controle de nomes de classes: Dentro de um pacote não pode haver duas classes com o mesmo nome, porém, em pacotes diferentes, isto é totalmente possível. Se assim não fosse, teríamos que nos certificar sempre que as classes que estamos desenvolvendo não possuem o mesmo nome de nenhuma classe em algum pacote que estejamos utilizando. Por exemplo, é possível definirmos uma classe chamada Vector pertencente ao pacote ACMEVideo. Se uma terceira classe X quiser utilizar a classe Vector do pacote util e a classe Vector do pacote ACMEVideo deverá

² Pacotes são distribuídos em arquivos com extensão .JAR e consistem nas APIs do Java.

especificar o nome completo dos pacotes na declaração das variáveis da seguinte forma:

```
class X {  
  
    java.util.Vector atributo1;  
    ACMEVideo.Vector atributo2;  
  
    ...  
}
```

2. Controle de acesso: Vamos recapitular os mecanismos de controle de acesso a métodos e atributos vistos no Capítulo 5:

- **public:** É o menos restritivo de todos. Atributos e métodos declarados como public em uma classe podem ser acessados pelos métodos da própria classe, por classes derivadas desta e por qualquer outra classe em qualquer outro pacote (veremos mais classes derivadas posteriormente).
- **protected:** Atributos e métodos definidos como protected são acessíveis pelos métodos da própria classe e pelas classes derivadas.
- **private:** É o mais restritivo. Atributos e métodos declarados como private só podem ser acessados pelos métodos da própria classe.

Quando nenhum modificador é definido (acesso do tipo “package”), os atributos e métodos podem ser acessados pelos métodos da própria classe, pelas classes derivadas e por qualquer outra classe dentro do mesmo pacote.

Além disso, existem os modificadores de acesso a classes. A declaração de uma classe geralmente começa com o modificador de acesso public, o que significa que pode ser acessada por qualquer outra classe. Se for omitido, a classe será visível apenas por outras classes do mesmo pacote que a contém.

7.2 – Atributos e Métodos de Classe

Variáveis de classe são declaradas de forma semelhante às variáveis de instância, com a diferença de que devemos acrescentar a palavra reservada **static**:

static tipo identificador [= valor];

Uma variável de classe é compartilhada por todas as instâncias desta classe. Por exemplo, considerem a classe X e o trecho de código que utiliza X:

```
class X {  
  
    public static int a = 10;  
    ...  
}
```

```
X var1 = new X();  
X var2 = new X();  
  
System.out.println(var1.a); //imprime 10  
var1.a = 20;  
System.out.println(var2.a); //imprime 20
```

Quando var1 modifica a variável de classe “a”, o novo valor é visível a todos os objetos desta classe, pois “a” representa a mesma posição de memória acessada por todos.

Variáveis de classe não podem ser inicializadas nos construtores. Se precisarem de inicialização, isto geralmente é feito no momento da declaração da variável.

Métodos de classe também são declarados com o auxílio do modificador **static**:

```
[modificador] static tipo_retorno identificador ([argumentos]) {
```

```
    //Corpo do método
```

```
}
```

Como um método de classe é compartilhado por todas as instâncias da classe, ele não se refere a um objeto em particular e por isso não recebe o ponteiro implícito *this*. Métodos de classe podem acessar somente variáveis de classe e outros métodos estáticos.

Uma característica interessante dos métodos estáticos, bem como dos atributos, é que eles podem ser acessados mesmo sem a existência de uma instância da classe.

```
class X {  
  
    public static int a = 10;  
  
    static void setA(int x){ a = x; }  
  
}
```

```
...  
X.setA(20);  
System.out.println(X.a);  
...
```

No exemplo acima, o método e atributo estático da classe X foram acessados diretamente referenciando-se a classe, sem a necessidade de criarmos uma instância da mesma .

Nas bibliotecas do Java encontramos inúmeros casos de atributos e métodos estáticos. Nosso conhecido método main(), por exemplo, é um método estático, que é chamado pela JVM quando uma aplicação é executada. A classe Math prove métodos de classe que executam várias funções matemáticas, como funções trigonométricas e logaritmos. Um exemplo é o método Math.sqrt() que calcula a raiz quadrada de um número não negativo. A classe System prove variáveis de classe para representar o estado do sistema inteiro. System.out é uma variável de classe que faz referência ao objeto PrintStream, que representa o fluxo padrão de saída. O método println() é um método de instância da classe PrintStream.

7.3 – Destrotores e o método finalize()

Em linguagens como C, uma classe pode ter um destrutor, utilizado normalmente para liberar recursos requisitados pelo objeto, como memória alocada, arquivos abertos, etc.

Java gerencia a memória automaticamente, então um objeto não precisa liberar qualquer memória alocada. Consequentemente, Java não suporta destrutores. Ao invés disso, para permitir a um objeto liberar outros recursos (diferentes de memória), tal como arquivos abertos, Java possibilita a implementação de um método **finalize()**.

O método `finalize()` é chamado automaticamente quando um objeto é coletado, portanto não é possível saber quando este será chamado. Ele não possui tipo de retorno e não assume nenhum parâmetro.

Exercícios :

- 1) Dada a classe abaixo, assinale as expressões que são corretas.

```
public class Filme {  
    private static float preco = 3.50f;  
    private String taxa;  
  
    public static void setPreco(float novoPreco)  
    {  
        preco = novoPreco;  
    }  
    public float getPreco() {  
        return preco; }  
}
```

```
Filme.setPreco(3.98f);  
Filme mov1 = new Filme();  
mov1.setPreco(3.98f);  
float a = Filme.getPreco();  
float b = mov1.getPreco();
```

- 2) Nesta lição iremos melhorar as classes da lição passada com os recursos que aprendemos aqui. A classe Cliente possui dois construtores: um sem argumento e outro que recebe o nome, endereço e telefone de um cliente como parâmetro.

- Crie uma variável estática e privada na classe Cliente do tipo inteiro e chamada *proximoId*. Esta deve ser inicializada com zero.
- Modifique o construtor default para que este incremente o valor desta variável e atribua o resultado para o atributo id do novo objeto.
- Modifique o outro construtor para que ele chame o construtor default e desta forma também inicialize o atributo id.

Obs: Em Java, para chamarmos um método construtor dentro de outro método qualquer, não podemos chamar o método pelo seu nome mas sim utilizando a sintaxe abaixo:

this (<lista de argumentos>);

- Recompile e execute novamente a aplicação. Observe que agora o valor de id não é mais zero.
 - Qual a vantagem de se usar um atributo de classe para fazer o controle do id dos objetos?
- 3) Faça as mesmas modificações para a classe Item, porém com a variável estática *proximoId* iniciando em 1000.

8 – Herança e Polimorfismo

Nas seções anteriores aprendemos sobre classes, objetos e encapsulamento, conceitos fundamentais do paradigma da orientação a objeto. Nesta seção, abordaremos dois outros conceitos importantes e relacionados entre si: **herança** e **polimorfismo**.

8.1 – Herança

A herança é a principal característica de distinção entre um sistema de programação orientado a objeto e outros sistemas de programação. As classes são inseridas em uma hierarquia de especializações de tal forma que uma classe mais especializada herda todas as propriedades da classe mais geral a qual é subordinada na hierarquia. A classe mais geral é denominada **superclasse** e a classe mais especializada **subclasse**.

O principal benefício da herança é a reutilização de código. A herança permite ao programador criar uma nova classe programando somente as diferenças existentes na subclasse em relação à superclasse. Isto se adequa bem a forma como compreendemos o mundo real, no qual conseguimos identificar naturalmente estas relações.

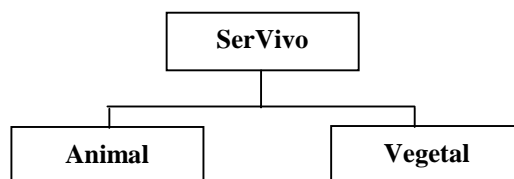
A fim de exemplificarmos este conceito, vamos considerar que queiramos modelar os seres vivos pluricelulares existentes no planeta. Podemos então começar com a classe SerVivo abaixo:



Ela modela as características que todo ser vivo deve possuir, como a capacidade de reproduzir-se ou a necessidade de alimentar-se. Sendo assim, a classe SerVivo define atributos e métodos tais como:

- Atributos: Alimentos, Idade
- Métodos: Nascer, Alimentar, Respirar, Crescer, Reproduzir, Morrer

Os seres vivos por sua vez classificam-se em Animais e Vegetais, os quais possuem características próprias que os distingue:



Analisando o problema em questão (o de modelar os seres vivos), nós naturalmente identificamos classes que são especializações de classes mais genéricas e o conceito de herança da orientação a objeto nos permite implementar tal situação.

Os animais e vegetais antes de tudo são seres vivos e cada subclasse herda automaticamente os atributos e métodos (respeitando as regras dos modificadores de acesso) da superclasse, neste caso, a classe SerVivo. Além disso, as subclasses podem prover atributos e métodos adicionais para representar suas próprias características. Por exemplo, a classe Animal poderia definir os seguintes métodos e atributos:

- Atributos: Forma de Locomoção, Habitat, Tempo Médio de Vida
- Métodos: Locomover

A herança na programação é obtida especificando-se qual superclasse a subclasse estende. Em Java isto é feito utilizando-se a palavra chave **extends**:

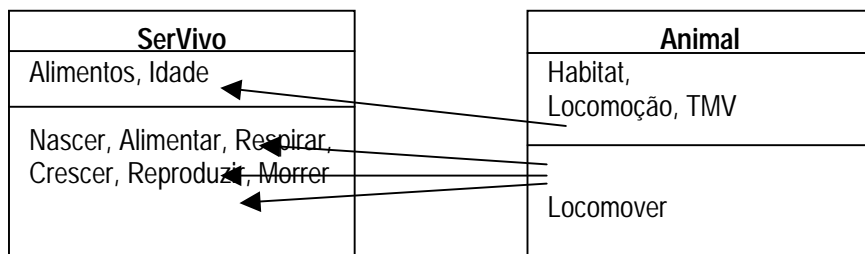
```
public class SerVivo {
    //Definição da classe SerVivo
}
public class Animal extends SerVivo {
    //Atributos e métodos adicionais que distinguem um Animal de um SerVivo
    //qualquer
}
```

Em Java, todas as classes, tanto as existentes nas APIs como as definidas pelos programadores, automaticamente derivam de uma superclasse padrão, a classe **Object**. Se uma classe não especifica explicitamente uma superclasse, como o caso da classe SerVivo, então podemos considerar que esta deriva diretamente de Object, como se ela tivesse sido definida como:

```
public class SerVivo extends Object { ... }
```

Além disso, Java permite apenas **herança simples**, isto é, uma classe pode estender apenas de uma **única** outra classe.

Resumindo, a classe SerVivo define os atributos e métodos que são comuns a qualquer tipo de ser vivo. A subclasse Animal herda estes métodos e atributos, já que Animal é um SerVivo, e tem que especificar apenas seus atributos e métodos específicos.



8.2 – Polimorfismo

O termo Polimorfismo origina-se do grego e quer dizer "o que possui várias formas". Em programação está relacionado à possibilidade de se usar o mesmo nome para métodos diferentes e à capacidade que o programa tem em discernir, dentre os métodos homônimos, aquele que deve ser executado. De maneira geral o polimorfismo permite a criação de programas mais claros, pois elimina a necessidade de darmos nomes diferentes para métodos que conceitualmente fazem a mesma coisa, e também programas mais flexíveis, pois facilita em muito a extensão dos mesmos.

O polimorfismo pode ser de duas formas, **estático** ou **dinâmico**:

- **Polimorfismo Estático:** Ocorre quando na definição de uma classe criamos métodos com o mesmo nome porém com argumentos diferentes. Dizemos neste caso que o método está **sobrecarregado** (*overloading*). A decisão de qual método chamar é tomada em tempo de compilação, baseada nos argumentos que foram passados.

Como exemplo de polimorfismo estático, podemos relembrar dos vários construtores que criamos para a classe Círculo no Capítulo 6:

```
Círculo() { }
```

```
Círculo(Ponto centro, Ponto extremidade, Cor c) {  
    raio = new Linha(centro, extremidade, c);  
    cor = c;  
}
```

```
Círculo (double x1, double y1, double x2, double y2, Cor c)  
{  
    raio = new Linha(new Ponto(x1, y1), new Ponto(x2, y2), c);  
    cor = c ;  
}
```

Todos estes métodos construtores possuem o mesmo nome mas devem ser diferidos entre si pelos parâmetros que recebem.

Quando num programa fazemos

```
Círculo circulo = new Círculo(linha);
```

o compilador consegue decidir em tempo de compilação qual método chamar, neste caso, o método Círculo que recebe uma linha como parâmetro.

- **Polimorfismo Dinâmico:** Esta associado com o conceito de herança e ocorre quando uma subclasse redefine um método existente na superclasse. Dizemos neste caso que o método foi **sobrescrito** (*overriding*) na subclasse. A decisão de qual método executar é tomada somente em tempo de execução, como veremos mais adiante.

O polimorfismo dinâmico ocorre quando uma subclasse redefine um método de sua superclasse a fim de prover ao método um comportamento mais adequado às suas características. Vamos rever a classe animal conforme a definimos acima

```
class Animal extends SerVivo{  
  
    String formaLocomoção;  
    String habitat;  
    int tempoMédioVida;  
  
    public void locomover() { ... }  
  
}
```

Como nem todo ser vivo nasce, cresce, alimenta-se, respira, se reproduz e morre da mesma maneira, é razoável que queiramos redefinir todos estes métodos na classe animal

```

class Animal extends SerVivo{

    String formaLocomoção;
    String habitat;
    int tempoMédioVida;

    public void locomover() { ... }
    public void nascer() { ... }
    public void crescer() { ... }
    public void alimentar() { ... }
    public void respirar() { ... }
    public Animal reproduzir() { ... }
    public morrer() { ... }

}

```

Os métodos na subclasse devem ser definidos com a mesma “assinatura” do método na superclasse, isto é, com o mesmo nome, tipo de retorno e argumentos.

Bem, mas a vantagem do polimorfismo dinâmico não é apenas a de permitir maior flexibilidade na modelagem das classes de objetos. Para entendermos o que há de mais fantástico nele, temos que nos atentar para o seguinte:

...

```

SerVivo x; —> Declara uma variável x do tipo SerVivo
Animal y = new Animal(); —> Declara e inicializa uma variável y do tipo Animal
x = y;

```

...

É extremamente comum e útil em OO atribuímos a uma variável de um tipo base um objeto de um tipo derivado direta ou indiretamente deste tipo base.

Tendo isto em mente, poderíamos definir um método da seguinte forma

```

void analisaSerVivo (SerVivo ser) {
    //Este método faz a análise clínica de qualquer SerVivo e para isso precisa pedir
    //ao animal que respire

        ser.respirar();
        ...
    }

```

e num determinado momento chamá-lo desta maneira

```

...
Animal animal = new Animal();
analisaSerVivo(animal);
...

```

Neste caso fica a questão, se o método tem um argumento declarado como SerVivo e recebe como parâmetro um objeto Animal, quando tiver que executar o método

respirar() qual método será efetivamente chamado, o método respirar definido em SerVivo ou o método respirar definido em Animal?

O método executado será o mais apropriado, isto é, aquele pertencente ao objeto que foi passado à variável, neste caso, o método respirar existente em Animal.

Isto é o polimorfismo dinâmico, que recebe este nome porque o compilador não consegue nestes casos decidir em tempo de compilação qual método chamar, já que uma variável de um tipo base pode receber qualquer tipo derivado. Esta decisão é feita apenas em tempo de execução, quando aí o Java poderá saber qual objeto a variável está de fato referenciando.

Se o polimorfismo dinâmico não existisse, seria necessário um método analisaSerVivo para cada ser vivo existente, e sempre que um ser vivo novo fosse acrescentado o código teria que ser modificado. Da forma como aqui está, o método continuará funcionando para qualquer ser vivo existente no projeto.

8.3 – Criação de um objeto de uma subclasse e o ponteiro *super*

Objetos são sempre construídos da classe mais alta na hierarquia em direção à mais baixa, isto é, da classe **Object** até a classe que está sendo instanciada com o operador **new**. Isto garante que uma subclasse pode sempre contar com a construção apropriada de sua superclasse.

Por padrão, os construtores default das superclasses (aqueles sem argumentos) é o que são chamados quando a superclasse é criada. Se quisermos mudar este comportamento devemos usar o ponteiro *super*.

O ponteiro *super* é uma referência para a superclasse e está disponível implicitamente em qualquer classe que possua um ancestral. Como vimos, os métodos na superclasse podem ser sobrescritos na subclasse através da criação de métodos na subclasse com a mesma assinatura do método da superclasse. Com o *super* é possível acessar um método específico da superclasse quando este foi sobrescrito pela subclasse. Por exemplo, vamos recordar a classe Animal definida acima que implementava sua própria versão de todos os métodos de sua superclasse, isto é, SerVivo:

```
class Animal extends SerVivo{

    String formaLocomoção;
    String habitat;
    int tempoMédioVida;

    public void locomover() { ... }
    public void nascer() { ... }
    public void crescer() { ... }
    public void alimentar() { ... }
    public void respirar() { ... }
    public Animal reproduzir() { ... }
    public morrer() { ... }
}
```

Vamos supor que num determinado momento quiséssemos acessar o método alimentar-se tal qual como este foi implementado na superclasse. Para isto devemos usar o ponteiro super da seguinte maneira:

```
...
super.alimentar();
...
```

| **Acessa o método alimentar-se de SerVivo.**

Bem, mas como afinal de contas podemos mudar o comportamento padrão de quando uma subclasse é instanciada os construtores default serem executados?

Um dos usos mais comuns da referência *super* é chamar um construtor de uma superclasse. Para isso, este deve ser acrescentado na **primeira linha** do construtor da subclasse. Por exemplo, se a classe SerVivo tiver um construtor que receba a idade do ser vivo como parâmetro

```
SerVivo(int idade){
    this.idade = idade;
}
```

e queremos que este construtor seja chamado quando a classe Animal for instanciada ao invés do construtor padrão SerVivo(), fazemos

```
Animal (int idade){
    super(idade);
    ...
}
```

8.4 – Métodos e Classes final

Vimos no Capítulo 3 que para criar variáveis que são constantes devemos usar o comando **final**. Vamos agora ver duas novas utilizações para ele:

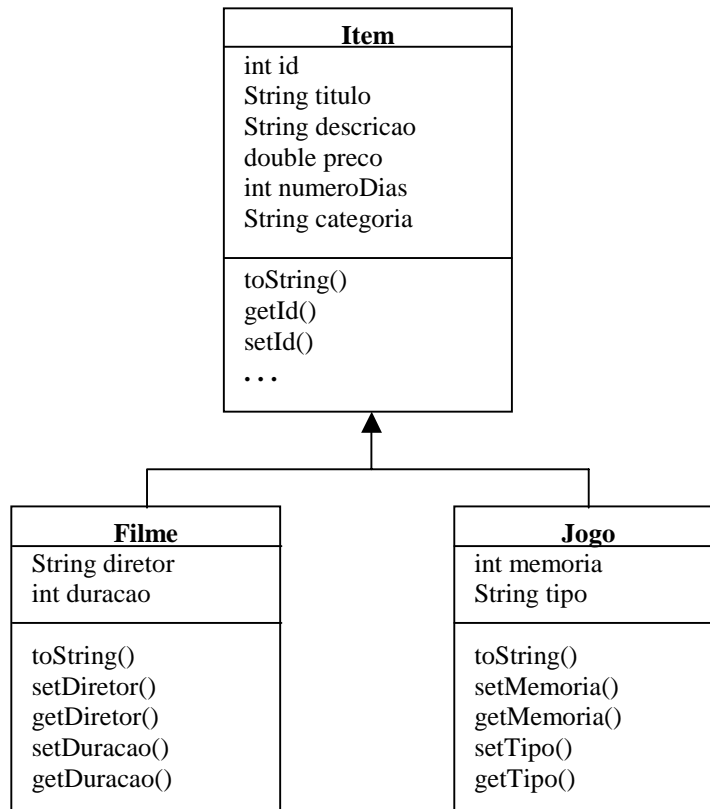
- Métodos final: Métodos definidos como final não podem ser sobreescritos. Geralmente isto é feito com o intuito de garantir segurança, prevenindo que métodos vitais, tais como aqueles que fazem autenticação de usuários, sejam redefinidos. Exemplo:

```
public final boolean checkPassword(String p) {
    ...
}
```

- Classes final: Uma classe declarada como final não poderá ser estendida por outra classe. Esta é uma forte decisão no projeto e significa que a classe é auto-suficiente para dispor de todos os seus requisitos atuais e futuros. Classes definidas como final podem ajudar o compilador a produzir códigos mais otimizados. Como uma classe final não pode ser estendida, quando o compilador encontra uma chamada a algum método desta classe ele não gerará o código necessário para execução do polimorfismo dinâmico, o qual acarreta um overhead durante a execução, mas sim fará um acoplamento estático com o método em questão.

Exercícios:

Nesta lição você vai adicionar duas novas classes que derivam de Item. A ACME Vídeo Locadora decidiu expandir seus negócios e começar a alugar jogos além dos vídeos. Como jogos possuem alguns atributos diferentes dos filmes, você decidiu criar subclasses para cada um destes tipos de itens. Cada item possui seus próprios métodos e redefinem o método toString() da classe Item. Abaixo segue o diagrama UML que mostra o relacionamento entre Item, Filme e Jogos. A seta é a notação UML para herança.



Abra o projeto ACME Video da lição 8.

- 1) Defina uma nova classe Filme.
 - a) Defina a classe Filme que estende a classe Item. A nova classe deve incluir os atributos e métodos definidos no diagrama acima.
 - b) Adicione um construtor que aceite a seguinte lista de atributos como argumentos: titulo, descricao, preco, numeroDias, categoria, diretor e duracao.
 - c) Inclua uma chamada ao construtor da superclasse Item usando a seguinte sintaxe

super(titulo, descricao, categoria , preco, numeroDias)

na primeira linha deste construtor.

- d) Rescreva o método toString() de forma que este retorne uma string com os valores concatenados de id, titulo, diretor e preco da seguinte maneira:

1001 (filme) A Vida é Bela (Leonardo Benine) – R\$ 6,00

O texto deve mostrar “(filme)” para indicar que este item é um filme.

2) Defina uma nova classe Jogo

- a) Defina a classe Jogo que estende a classe Item. A nova classe deve incluir os atributos e métodos definidos no diagrama acima.
- b) Acrescente um construtor que aceite como argumentos a seguinte lista de atributos: titulo, descricao, categoria, preco, numeroDias, memoria, tipo
- c) Adicione uma chamada ao construtor da superclasse Item usando a seguinte sintaxe

super(titulo, descricao, categoria, preco , numeroDias)

na primeira linha deste construtor.

- d) Rescreva o método toString() de forma que este retorne uma string com os valores concatenados de id, titulo, tipo e preco da seguinte maneira

1011 (jogo) Lifelike racing (Play Station) - R\$ 5,00

O texto deve mostrar “(jogo)” para indicar que este item é um jogo.

- 3) Teste a aplicação. No método main() do projeto, declare duas variáveis do tipo Item chamadas filme e jogo respectivamente. Inicialize-as chamando os construtores recém criados da seguinte maneira:

Item filme = **new** Filme(...);

Item jogo = **new** Jogo(...);

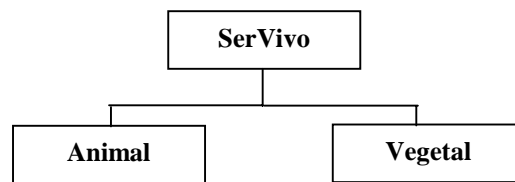
Chame o método toString() dos dois objetos. Perceba como a atuação do polimorfismo dinâmico irá garantir a correta execução do método apropriado da subclasse.

9 – Estruturando o Código com Classes Abstratas e Interfaces

Quando a hierarquia de classes em um programa orientado a objetos torna-se complexa, as classes mais acima (raiz e proximidades) costumam ser bastante gerais. Passa a ser interessante então, poder definir “protótipos” de classes que serão usadas para guiar a criação de outras classes. Elas implementam ou apenas definem comportamentos que são genéricos o suficiente para serem comuns a várias outras classes.

9.1 – Classes Abstratas

Uma maneira de criar este protótipo é definindo **classes abstratas**. Uma classe abstrata é uma classe que **não** pode ser instanciada. Vejamos nossa hierarquia de classes para modelar seres vivos:



A classe SerVivo pode ser declarada como abstrata pois não possui detalhes o suficiente para representar sozinha um objeto completo. O usuário não poderá criar objetos da classe SerVivo pois esta é apenas uma classe parcial e existe apenas para ser estendida por subclasses mais especializadas, tal como Animal e Vegetal.

```
public abstract class SerVivo {  
  
    String Alimentos;  
    int Idade;  
    public void locomover() { ... }  
    public void nascer() { ... }  
    public void crescer() { ... }  
    public void alimentar() { ... }  
    public void respirar() { ... }  
    public SerVivo reproduzir() { ... }  
    public morrer() { ... }  
  
}
```

Utilizamos a palavra reservada **abstract** para definir uma classe abstrata. Se tentarmos instanciar uma classe abstrata receberemos um erro de compilação. Uma classe abstrata pode conter:

- Métodos e atributos como em classes convencionais. As classes derivadas de uma classe abstrata herdam da mesma forma os métodos e atributos. Estes métodos podem ser sobrescritos nas subclasses mas isto não é obrigatório.
- Métodos abstratos, que obrigatoriamente devem ser sobrescritos pelas subclasses.

Métodos abstratos podem existir apenas em classes abstratas. Como uma classe abstrata está num nível muito alto de abstração, muito freqüentemente não possui detalhes o suficiente para implementar determinados comportamentos que serão comuns a todas as suas subclasses. Ao invés disso, elas apenas definem quais serão estes comportamentos obrigatórios através da criação de métodos abstratos.

Um método abstrato obrigatoriamente deve ser implementado em qualquer uma das subclasses que estendem a superclasse abstrata. A classe `SerVivo` por exemplo define uma série de métodos que ela por si só não é capaz de prever qual será o comportamento para todas as subclasses mas que ao mesmo tempo são comportamentos comuns a todos os seres vivos. Sendo assim, estes métodos devem ser declarados como abstratos da seguinte forma:

```
public abstract class SerVivo {  
  
    String Alimentos;  
    int Idade;  
    public abstract void locomover();  
    public abstract void nascer();  
    public abstract void crescer();  
    public abstract void alimentar();  
    public abstract void respirar();  
    public abstract SerVivo reproduzir();  
    public abstract void morrer();  
  
}
```

Todos os diferentes tipos de seres vivos executam estas operações de diferentes maneiras.

Para definir um método abstrato devemos prover apenas a assinatura do método, isto é, seu nome, argumentos e tipo de retorno. A implementação deste ficará a cargo de cada subclasse concreta, que poderá dar a sua versão para ele.

A utilização de classes e métodos abstratos é uma forma muito comum para criarmos e utilizarmos bibliotecas orientadas a objetos: acoplamos nossas classes definindo heranças e implementamos os métodos abstratos, os quais garantem que estas classes terão garantidamente um protótipo básico.

9.2 – Interfaces

Uma interface é semelhante a uma classe abstrata exceto pelo fato desta não poder ter qualquer método concreto ou variável de instância. Uma interface representa uma coleção de declarações de métodos abstratos e possivelmente constantes.

Qualquer classe que implementa uma interface deve obrigatoriamente implementar todos os métodos especificados pela interface. Assim, interfaces fornecem a idéia de um “contrato” que qualquer subclasse deve obedecer. Outra característica importante é que uma classe pode derivar de apenas uma única outra classe (herança simples), mas pode implementar um número indeterminado de interfaces. Por exemplo, a classe `Animal` poderia ser definida da seguinte forma:

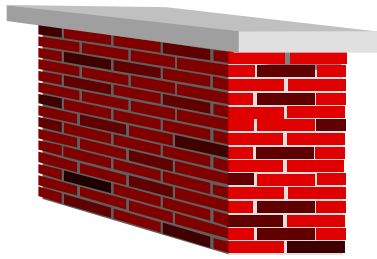
```
class Animal extends SerVivo implements Sortable, Listable {  
  
    ...  
  
}
```

Uma classe pode implementar mais de uma interface.

Como mostrado anteriormente, `Animal` herda de `SerVivo` seus dados e comportamentos e como `SerVivo` possui métodos abstratos, estes devem ser implementados em `Animal`.

Além disso, como fizemos `Animal` implementar as interfaces `Sortable` e `Listable`, este deve implementar também todos os métodos especificados nestas interfaces.

Uma interface descreve um aspecto de comportamento que muitas classes diferentes podem requerer. O nome de uma interface é freqüentemente um adjetivo como `Sortable` (ordenável) ou `Steerable` (algo que pode ser pilotado). A interface `Steerable` pode incluir métodos tais como `turnRight()`, `turnLeft()`, `turnAround()`, etc. Qualquer classe que precisar demonstrar o comportamento de poder ser dirigida pode implementar a interface `Steerable`.



Não pilotável



Pilotável

É importante notar que classes que implementam a mesma interface podem não ter absolutamente nada a ver umas com as outras (o que não ocorre quando classes implementam herança em relação à mesma superclasse), a não ser o fato de apresentarem comportamentos semelhantes, porém, que podem estar implementados de formas completamente diferentes. Por exemplo, os principais pacotes Java incluem inúmeras interfaces padrão, tais como `Runnable`, `Cloneable`, dentre outras. Estas interfaces são implementadas por todos os tipos de classes que não têm nada em comum, exceto o fato de precisarem ser “clonáveis” ou “executáveis”.

Para definir uma interface devemos usar a palavra reservada **interface**. Todos os métodos especificados são implicitamente públicos e abstratos e qualquer variável é implicitamente `public static final`, ou seja, são constantes. A interface `Steerable` por exemplo, poderia ser definida da seguinte maneira

```
public interface Steerable{  
  
    int MAXTURN = 45;  
    void turnLeft (int deg);  
    void turnRight (int deg);  
}
```

o que equivale a seguinte definição se não suprimirmos o `public`, `static`, `final` e `abstract`

```
public interface Steerable{  
  
    public static final int MAXTURN = 45;  
    public abstract void turnLeft (int deg);  
    public abstract void turnRight (int deg);  
}
```

Uma interface pode ser definida de qualquer uma das maneiras mas é mais comum encontrarmos da primeira forma.

Para implementarmos uma interface usamos a palavra reservada **implements**. Como exemplo, vamos considerar a classe Iate abaixo:

```
public class Iate extends Barco implements
Steerable {

    public void turnLeft (int deg){ ... }
    public void turnRight (int deg) { ... }

}
```

Neste exemplo, uma classe Iate que deriva da classe Barco tem a característica de precisar ser pilotada e, sendo assim, implementa a interface Steerable. A classe Iate deve portanto, implementar todos os métodos da interface, neste caso turnLeft(int deg) e turnRight(int deg).

Um exemplo real - Ordenação

A ordenação é um exemplo clássico do uso de uma interface. Ordenar uma lista de objetos é um processo bem definido que não precisa ser escrito repetidas vezes e muitas classes diferentes podem requerer serem “ordenáveis”. Assim, uma rotina de ordenação precisa possuir a habilidade de ordenar qualquer objeto seguindo as características particulares de cada um. A abordagem tradicional de programação poderia propor a reescrever a rotina de ordenação para cada tipo diferente de objeto que precisasse ser ordenado. Usando a programação orientada a objetos de forma adequada pode-se eliminar esta necessidade e toda dificuldade de manutenção que ela acarreta.

“Vamos supor que queiramos que seja possível ordenar um conjunto de objetos da classe Animal por ordem de idade.”

O que devemos fazer, correr atrás de um algoritmo de ordenação e implementar um método que seja capaz de ordenar um vetor de animais? Graças a orientação a objetos e ao Java a resposta é não! Java possui a classe java.util.Arrays que possui um método estático sort(Object[]) que realiza a ordenação de um array qualquer. Bem, mas como este método consegue a façanha de ordenar um array de objetos arbitrários? Se ele não conhece o objeto, como vai saber como comparar um objeto do vetor com outro de forma a determinar a ordem relativa entre eles?

É aí que entra a necessidade de uma interface. A interface Comparable especifica métodos requeridos para fazer a ordenação funcionar em cada tipo de objeto que precisa ser ordenado. Cada classe implementa a interface baseada em suas necessidades específicas de ordenação e apenas ela precisa saber como seus objetos devem ser comparados. Implementar a ordenação de maneira OO provê um modelo muito bem adequado para reutilização de código. O código da ordenação é completamente isolado dos objetos que usam a ordenação.

Voltando ao exemplo, as classes e interfaces a serem utilizadas para ordenar um conjunto de Animais são:

Criadas pelos especialistas em ordenação

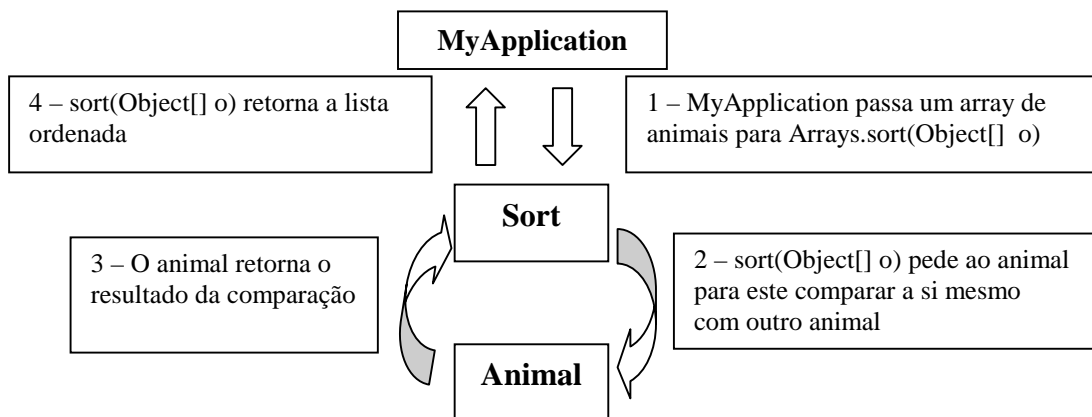
```
public interface Comparable
public class Arrays
```

Criadas pelos especialistas em modelar seres vivos

```
public class Animal
    implements Comparable
public class
    MyApplication
```

- A interface Comparable declara um único método: compareTo(Object). Este método deve ser implementado por qualquer classe que precisa utilizar os métodos da classe Arrays.
- A classe Arrays contém o método estático sort(Object[]) para ordenar um array de objetos. A maioria dos algoritmos de ordenação funcionam através da comparação de pares de objetos. O método sort(Object[]) faz esta comparação chamando o método compareTo(Object) do objeto no array.
- A classe Animal implementa a interface Comparable. Ela contém um método compareTo(Object) que compara o animal corrente com o animal passado como argumento do método.
- MyApplication simboliza qualquer aplicação que precisa ordenar uma lista de animais.

Esquema de Execução



Interface Sortable

```
public interface Comparable {
    //compareTo(Object): Compara este objeto
    //com o outro passado como argumento
    // Retorno:
    // 0 se este objeto for igual a obj2
    // um valor < 0 se este objeto for < obj2
    // um valor > 0 se este objeto for > obj2
    int compareTo (Object obj2);
}
```

A classe Arrays

```
public class Arrays{
    public static void sort (Object[] items){
        for (int i = 1; i < items.length; i++){
            for (int j = 0; j < items.length -1; j++){
                if (items[j].compareTo(items[j+1]) > 0){
                    Comparable tempitem = items[j+1];
                    items[j+1] = items[j];
                    items[j] = tempitem;
                } } } } }
```

Uma interface também define um tipo.

Quando o método sort precisa comparar dois itens do array ele chama o método compareTo de um dos itens e passa o outro como argumento.

Note que o método sort não sabe nada sobre o tipo de objeto que ele está ordenando. Ele sabe apenas que estes são objetos que implementam a interface Comparable e, sendo assim, ele pode contar com a existência do método compareTo nestes objetos.

Podemos considerar uma interface como um contrato entre o objeto que usa a interface e o objeto que a implementa:

- A classe Animal (a implementadora) concorda em implementar um método chamado compareTo(Object), com parâmetros e valor de retorno que seguem o especificado pela interface.
- A classe Arrays (usuário) concorda em ordenar uma lista de objetos na ordem correta fazendo valer-se do método compareTo.

A classe Animal

```
public class Animal extends SerVivo
    implements Comparable{

    public int compareTo(Object animal2){
        int idade1 = this.idade;
        int idade2 = ((Animal)animal2).idade;
        if (idade1 < idade2) return (-1);
        if (idade1 > idade2) return (1);
        if (idade1 == idade2) return (0);
    }
}
```

Foi necessário fazer um cast pois o tipo do argumento é Object.

O método compareTo(Object) pode comparar os dois objetos de qualquer maneira que quisermos, contanto que ele retorne um inteiro para indicar a ordem relativa entre eles.

Por fim, a classe MyApplication para ordenar um array de Animais, pode simplesmente chamar **Arrays.sort(Object[])** passando o array de objetos a ser ordenado.

10 – Usando Strings e Arrays

Ao término deste capítulo você estará apto a:

- Criar strings em Java
- Realizando Operações em Strings
- Converter primitivos para Strings e vice-versa
- Manipular strings usando a classe StringBuffer
- Criar arrays de elementos

10.1 – Strings

Como em muitas linguagens de programação, strings são usadas extensivamente em programas Java o que fez com que a API do Java disponibiliza-se a classe String para ajudar na manipulação deste tipo de dado. Ela faz parte do pacote java.lang, o qual é automaticamente importado em todas as classes java.

A classe String representa uma string imutável. Isto significa que, uma vez criado um objeto desta classe, você não poderá alterá-lo. Se você quer modificar o conteúdo de uma String, você deve usar a classe StringBuffer, a qual foi desenvolvida para este propósito.

Existem várias maneiras de se criar uma String:

- Atribuindo uma constante string a uma variável

```
String categoria = "Ação";
```

- Concatenando outras strings

```
String nome = primeiroNome + " " + ultimoNome;
```

- Usando o construtor

```
String nome = new String("Joe Smith");
```

Java utiliza o operador + para concatenação de strings. O método concat() na classe String é uma outra forma de fazer isso. O código seguinte produz strings equivalentes:

```
String nome = "San Wong";  
String nome = "Sam " + "Wong";  
String nome = "Sam ".concat("Wong");
```

É possível também concatenar qualquer tipo primitivo com uma string, pois este é implicitamente convertido para uma.

```
int idade = 25;  
System.out.println("Idade = " + idade);
```

10.1.1 – Operações em strings

A classe String disponibiliza inúmeros métodos para realizarmos operações sobre ela. Todas as operações **não** modificam a string original. Alguns delas são:

- Retornando o tamanho da string: **int length()**

```
String categoria = "Comedia";  
int len = categoria.length();
```

⇒ **len = 7**

- Encontrando um caracter numa posição específica: **char charAt(int index)**

```
String categoria = "Comedia";  
char c = categoria.charAt(1);
```

⇒ **c = "o"**

O índice começa em 0 (zero).

- Retornando uma substring: **String substring (int beginIndex, int endIndex)**

```
String categoria = "Comedia";  
String sub = categoria.substring(2,4)
```

⇒ **sub = "me"**

Este método retorna a substring que inicia em beginIndex e que termina em endIndex-1.

- Convertendo para upper e lowercase

```
String toUpperCase()  
String toLowerCase()
```

- Eliminando espaços em branco: **String trim()**

```
String comEspaco = " XXX ";  
String semEspaco = comEspaco.trim();
```

⇒ **semEspaco = "XXX"**

Retira os espaços das extremidades.

- Encontrando o índice de uma substring

```
String str = "XXXYXXY";  
int i = str.indexOf("Y");
```

⇒ **i = 3**

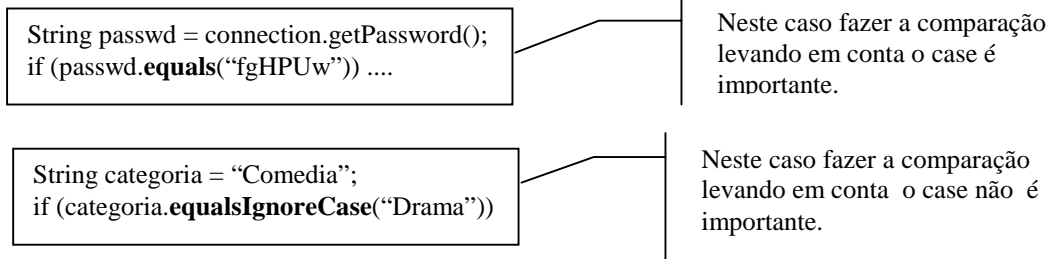
```
String str = "XXXYXXY";  
int i = str.lastIndexOf("Y");
```

⇒ **i = 6**

String indexOf (String) retorna o índice de uma string especificada.

String lastIndexOf (String) retorna a última ocorrência da string especificada.

- Comparando Strings : **boolean equals()** e **boolean equalsIgnoreCase()**



Os métodos retornam **true** se a string especificada contém exatamente o mesmo texto.

10.1.2 – Convertendo primitivos para Strings e vice-versa

A classe `String` possui um método estático chamado **valueOf()** que retorna uma representação em string de um tipo primitivo. Existe uma versão do método para cada tipo primitivo.

```
String sete = String.valueOf(7);
String umPontoZero = String.valueOf(1.0f);
```

O método `System.out.println()` também possui uma versão par cada tipo primitivo. Desta forma, é possível fazer

```
int count = 10;
System.out.println(count);
```

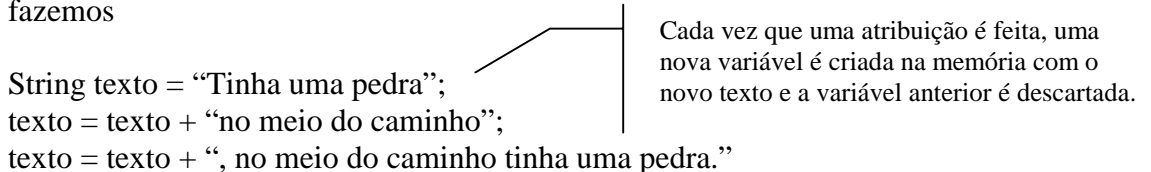
sem a necessidade de antes concatenar `count` com uma string (o que faria com que ele fosse automaticamente convertido para uma string).

É possível também converter uma string para um tipo primitivo através das classes de **wrapper**³. Para cada tipo primitivo Java provê uma classe de wrapper correspondente, a qual permite que um tipo primitivo seja tratado como um objeto. Cada classe de wrapper disponibiliza um método estático para converter uma `String` para o tipo primitivo correspondente. Exemplo :

```
int valor = Integer.parseInt("10");
float valor = Float.parseFloat("10.00");
```

10.1.3 – A classe `StringBuffer`

A classe `StringBuffer` representa strings que podem ser modificadas em tempo de execução. Assim, se você pretende fazer modificações numa string (acrescentar caracteres, remover caracteres, etc), você deve usar esta classe. Por exemplo, quando fazemos



³ A tradução de wrapper é invólucro, envoltório.

Esta abordagem para modificar o texto de uma String não é a mais eficiente, tendo em vista que novas variáveis são criadas em cada atribuição. O ideal é utilizar a classe StringBuffer e seu método append:

```
StringBuffer texto = new StringBuffer("Tinha uma pedra");  
texto.append(" no meio do caminho");  
texto.append(", no meio do caminho tinha uma pedra.");
```

10.2 – Arrays

Um array é útil quando queremos agrupar objetos do mesmo tipo. Possui as seguintes características:

- Todos os elementos devem ser do mesmo tipo
- Cada elemento pode armazenar um único item
- Os itens podem ser primitivos ou referências a objetos
- O tamanho do array é fixado quando de sua criação

10.2.1 – Arrays de Primitivos

Para criar um array de primitivos fazemos:

1 – Declaramos o array:

```
type[] arrayName;  
or  
type arrayName[];
```

Exemplo: `int[] powers;`

powers aponta para **null** até que seja inicializado com o operador **new**.

2 – Criamos efetivamente o array:

```
int[] powers;  
powers = new int[4];
```

Cada elemento é inicializado com o valor 0.

Um array deve ser criado com o uso do operador new e seu tamanho deve ser fixado dentro dos colchetes. O tamanho deve ser um inteiro mas não precisa ser uma constante, pode ser também uma expressão que retorna um inteiro.

Quando um array de primitivos é criado, **seus elementos são inicializados automaticamente** com valores padrão da seguinte forma:

- char : ‘\u0000’ (Valor Unicode para 0000)
- byte, short, int e long : 0
- boolean : false
- float e double: 0.0

3 – Inicializando os elementos do array. Duas formas:

• Atribuindo valores para cada elemento do array:

```
arrayName[index] = value;
```

Exemplo: `powers[0] = 1;`

O índice de um array inicia em 0.

- Usando inicializadores: `type[] arrayName = {value list};`

Exemplo: `int [] primes = {2,3,5,7};`

Não há a necessidade de usar o operador **new** nem de definir o tamanho do .

10.2.2 – Arrays de Referências

Os passos para criar um array de referências de objetos são os mesmos do array de primitivos, com uma exceção: não existe uma inicialização default para os elementos.

1 – Declaração:

`String[] categorias;`

categorias aponta para **null**.

2 – Criação:

`categorias = new String[3];`

Os elementos não recebem uma inicialização default. Cada item aponta inicialmente para **null**.

3 – Inicialização. Duas formas:

- Atribuindo valores para cada elemento

```
String[] arr = new String[4];
for (int i = 0; i < arr.length; i++){
    arr[i] = new String();
}
```

Todo array tem o atributo `length` que contém o número de elementos do array.

- Através de inicializadores

```
String[] categorias = {
    "Ação", "Comédia", "Drama" };
```

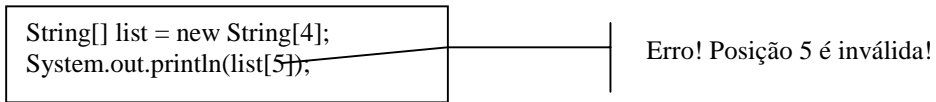
10.2.3 – Arrays como objetos

Quando criamos um array tanto de primitivos como de referências usamos o operador `new`, o qual nos retorna o endereço da primeira posição do array. Quando passamos um array como argumento para um método este é passado por referência, sendo assim, se o método alterar o conteúdo do array, estas mudanças dar-se-ão no array original e não numa cópia.

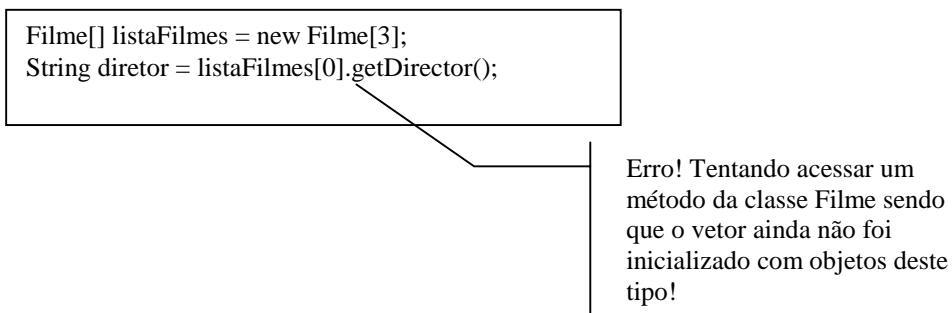
10.2.4 – Arrays e Exceções

Um dos motivos de dizermos que Java é uma linguagem robusta e segura é devido a forma de como ela trata de acessos ilegais em arrays. Iremos estudar exceções com mais detalhes no próximo capítulo, porém vamos aproveitar para dar uma olhada em duas exceções relacionadas a arrays.

- A primeira trata-se da **ArrayIndexOutOfBoundsException**, que ocorre quando tenta-se acessar uma posição inválida do array:

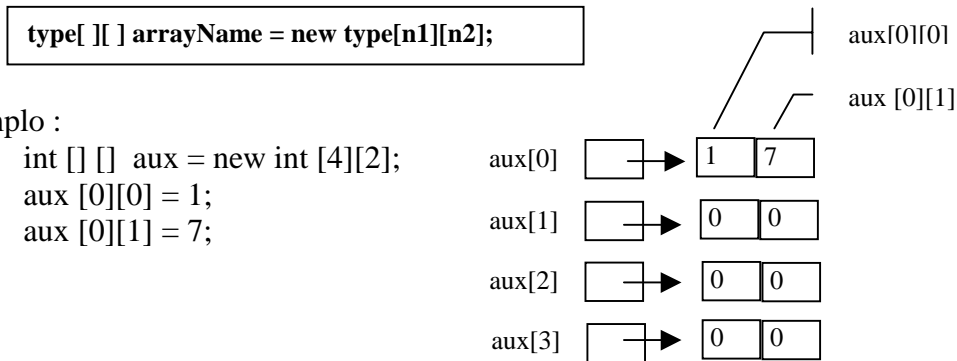


- A segunda, **NullPointerException**, ocorre quando tenta-se acessar um elemento que ainda não foi inicializado



10.2.5 – Arrays multidimensionais

Java suporta arrays multidimensionais, isto é, arrays de arrays.



Representação em Memória de um Array

Exercícios:

Abra a lição 10 e selecione o projeto ACMEVideo. Vamos realizar várias modificações na classe a fim de por em prática o que aprendemos nesta lição.

- 1) Faça as seguintes modificações na classe ACMEVideo.
 - a) Defina um atributo array *private static* de objetos Cliente chamado clientes com capacidade para armazenar seis objetos Cliente. Para economizar tempo, o código para criar um array de itens já foi adicionado para você.
 - b) Adicione uma variável de instância booleana, privada e estática chamada clientesConstruidos que serve para controlar se o array de clientes já foi inicializado. Inicialize esta variável com *false*.
 - c) Crie um método público e estático chamado constróiClientes(). Este método contem o código necessário para inicializar as seis posições do array de clientes (lembre-se que o construtor de cliente espera receber um nome, endereço e telefone. Haja imaginação!!). Uma inicialização deve parecer-se com isso:

```
clientes[0] = new Cliente("XXXX", "YYYY", "ZZZZ");
```

- d) Sete a variável clientesConstruidos para true no interior deste método.
 - e) Adicione agora um método público e estático que recebe o ID de um cliente como argumento, busca no array clientes por um cliente que tenha este ID e, caso encontre-o, retorna o objeto correspondente. O método pode usar um laço for para fazer a busca no array (você pode usar a propriedade array.length para saber o final do array) e deve se chamar buscaClientePorId(). Se um cliente não for encontrado o método deve retornar null. Adicione uma condição no método para verificar, antes de iniciar a busca, se existem clientes cadastrados (através da variável clientesConstruidos). Caso ainda não, faça o método inicializar o array chamando o método constróiClientes. O código irá se assemelhar a isto:

```
if (!clientesConstruidos) { constróiClientes(); }  
//o laço do for deve entrar aqui
```

Nessa altura do campeonato, já temos em nosso projeto de locadora quatro classes : a classe **Cliente** que manipula os clientes da loja, a classe **Item** que manipula informações gerais sobre os itens que podem ser alugados, a classe **Filme** e a classe **Jogo** que são tipos específicos de itens alugáveis. Para concluir o diagrama apresentado no capítulo 6, faltam as classes **ItemAlugado** e **Aluguel**.

- 2) Vamos definir agora a classe ItemAlugado, a qual controla as informações de um item quando este é alugado para um cliente.
 - a) Crie a classe ItemAlugado e inclua as seguintes variáveis privadas (não esqueça dos métodos getXXs e setXXs):

```
Item item  
String dataDevolução  
String dataDevolvida
```

Faça um import no pacote java.util.*

b) Adicione um método `getPreco()`. Como preço não é um atributo de `ItemAlugado`, você deve chamar o método `getPrice` do `Item`. Este é apenas um método de conveniência para sabermos a partir de `ItemAlugado` qual o preço de aluguel do item. O código deve assemelhar-se a isso:

```
public double getPreco(){
    return item.getPrice();
}
```

c) Crie um construtor para classe que aceite o ID de um `Item` (`int`) como parâmetro. O construtor deve usar este valor para chamar o método `ACMEVideo.buscaItemPorId()` (que já foi construído para você como um método da classe `ACMEVideo`), o qual retorna um objeto do tipo `Item`, e inicializar seu atributo `item`. O construtor também deve inicializar a variável `dataDevolucao` com o valor retornado pelo método `calcDataDevolucao()`;

3) Neste momento você está pronto para criar a última classe necessária para concluir a aplicação. A classe `Aluguel` representa uma locação efetivada. Deve controlar por exemplo qual cliente realizou a locação, os itens locados e a data que o empréstimo foi feito.

a) Crie uma nova classe `Aluguel` e inclua um `import` no pacote `java.util.*`.

b) Adicione as seguintes variáveis privadas:

```
ItemAlugado[] itens;
Cliente cliente;
int numeroAluguel;
Date dataAluguel;
int itemCount;
```

c) Adicione as seguintes variáveis privadas e estáticas:

```
static int proximoId = 500;
final static int MAXITENS = 10;
```

d) Crie um construtor que inicialize as três variáveis. Ele deve gerar o próximo número de aluguel baseado na variável estática `proximoId` (lembra como fizemos no capítulo 7 quando estudamos variáveis estáticas). Deve receber o ID de um cliente como parâmetro e utilizar o método `ACMEVideo.buscaClientePorId()` para inicializar a variável `cliente`. Por fim, deve criar o array `itens` com o máximo de `MAXITENS` posições. Você mesmo pode fazer este código ou usar o abaixo:

```
public Aluguel (int clienteId) {
    numeroAluguel = ++proximoId;
    cliente = ACMEVideo1.buscaClientePorId(clienteId);
    itens = new ItemAlugado[MAXITENS];
}
```

e) Crie um método chamado `addItem()` para adicionar um item alugado num aluguel. Use o código abaixo como guia.

```

public void addItem (int itemId) {
    ItemAlugado item = new ItemAlugado(itemId);
    // Adicione este item no array. Use a variável itemCount para saber qual a
    //posição
    ...
    //Incremente itemCount
}

```

- f) Por fim, você deve criar o método `imprimeRelatorio()` que imprime um relatório relacionado a um aluguel. O relatório deve ter o seguinte layout:

Nome do Cliente:

Telefone do Cliente:

Data do Aluguel:

Itens Alugados:

Item 1 : <Titulo> <Data Devolucao> <Valor>

...

Item n : <Titulo> <Data Devolucao> <Valor>

Custo Total:

- 4) Teste a aplicação. No método `main` do projeto `ACMEVideo` simule um aluguel da seguinte forma:

```

//crie um novo aluguel para o cliente 3
Aluguel aluguel = new Aluguel(3);
//adicione alguns itens locados
aluguel.addItem(1001);
aluguel.addItem(1002);
aluguel.addItem(1005);
//Imprima o relaorio
aluguel.imprimeRelatorio();

```

Você pode testar também para outras entradas. Clientes válidos variam de 1-6 e Itens válidos de 1001-1011.

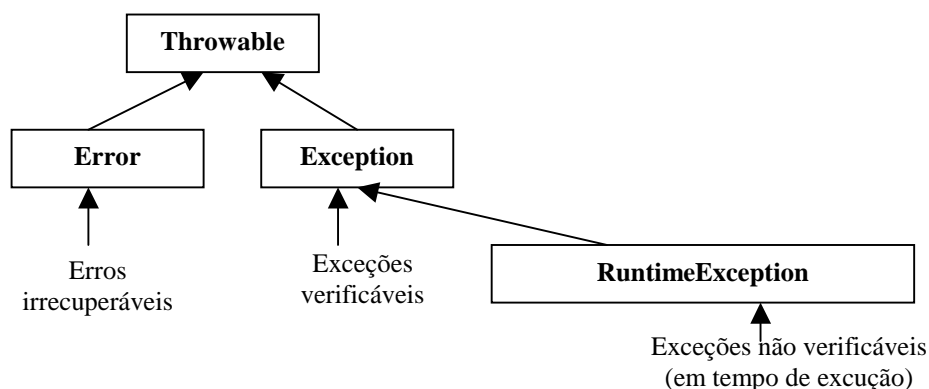
11 – Tratamento de Exceções

Uma **exceção** é um evento que ocorre durante a execução de um programa e que desfaz o fluxo normal de instruções. Por exemplo, tentar acessar um elemento fora dos limites de um array, tentar dividir um número por zero, são exemplos de exceções. Um **erro** em Java é uma condição anormal irreversível. Por exemplo, uma condição de erro ocorre se existe algum erro interno na JVM ou se a JVM fica sem memória. A diferença entre um erro e uma exceção é que uma exceção pode ser capturada e o programa pode seguir em frente a partir dali, já quando um erro ocorre o programa necessariamente irá terminar.

Quando uma exceção ocorre no interior de um método Java, o método cria um objeto do tipo Exception e o retorna para o sistema. Este processo é chamado de “disparar uma exceção”. O objeto Exception contém informações sobre a exceção, incluindo seu tipo e o estado do programa quando a exceção ocorreu. Quando uma exceção é disparada, o sistema busca em seqüência na pilha todos os métodos para tentar encontrar um capaz de tratar este tipo de exceção. Na terminologia Java o método “captura a exceção”. Se o sistema não encontra nenhum método apto a capturar a exceção, o programa inteiro termina.

11.1 – Classes de Exceções

Todos os erros e exceções em Java derivam da classe Throwable, resultando na hierarquia abaixo:



- **Erros:** Erros são extensões da classe Error. Se um erro é gerado, ele normalmente indica um problema que será fatal para o programa.
- **Exceções não verificáveis:** Este tipo de exceção deriva da classe RuntimeException. Podemos escolher o que fazer com uma exceção que ocorre em tempo de execução (como dividir um número por zero ou acessar um elemento inválido de um array); podemos tentar capturá-la e tratá-la ou então a ignorarmos. Se optarmos pela segunda opção, a JVM irá terminar o programa e imprimir o nome da exceção e a pilha de mensagens.
- **Exceção verificáveis:** Exceções deste tipo são extensões da classe Exception. Elas devem obrigatoriamente ser capturadas em algum lugar da aplicação, sob pena de ocorrer erro de compilação caso isso não seja feito.

11.2 – Tratando de Exceções

Quando chamamos um método que pode retornar uma exceção (tanto verificável quanto não), temos três alternativas:

1. Capturar a exceção e tratá-la
2. Deixar a exceção passar pelo método; em algum outro lugar ela deve ser capturada.
3. Capturar a exceção e disparar uma exceção diferente. A nova exceção deve ser capturada em algum lugar.

11.2.1 – Capturando Exceções

Se um bloco de código chama um ou mais métodos que podem disparar exceções, coloque o código dentro de um bloco *try*, com um ou mais blocos *catch* imediatamente posterior a este. Cada bloco *catch* trata de uma exceção particular. Pode-se opcionalmente incluir um bloco *finally* após todos os blocos *catch*, o qual é sempre executado mesmo que nenhuma exceção ocorra.

```
try {  
    //código que pode causar uma exceção  
}  
catch (exception1) {  
    //tratamento da exceção 1  
}  
catch (exception2) {  
    //tratamento da exceção 2  
}  
...  
finally {  
    //qualquer processamento final  
}
```

Para saber se um método de alguma API Java dispara uma exceção, consulte o JavaDoc.

Exemplo 1 – Capturando uma única exceção:

```
int num;  
String s = getNumFromForm();  
try{  
    //Pode disparar NumberFormatException  
    num = Integer.parseInt(s);  
}  
catch (NumberFormatException e) {  
    //trata a exceção  
}
```


Exemplo 2 – Capturando Múltiplas Exceções:

```
try {
//pode disparar MalformedURLException
URL u = new URL(str);
//pode disparar IOException
URLConnection c = u.openConnection();
}
catch (MalformedURLException e) {
System.out.println("Impossível abrir URL:" + e);
}
catch (IOException e) {
System.out.println("Não foi possível conectar:" + e);
}
```

Um bloco catch irá capturar a exceção especificada bem como qualquer uma de suas subclasses. Por exemplo, a classe `MalformedURLException` estende `IOException`, sendo assim, poderíamos substituir os dois blocos catch acima por

```
catch (IOException e) {

    System.err.println("Operação falhou:" + e);

}
```

Isso é útil quando iremos dar o mesmo tratamento para qualquer uma das exceções de uma mesma hierarquia.

Caso queiramos realmente tratar de forma diferente exceções de uma mesma hierarquia, devemos nos atentar para a ordem em que estas serão especificadas no blocos catch. **Ocorrerá um erro de compilação caso especifiquemos primeiro a superclasse e depois a subclasse da exceção.**

```
try {
//pode disparar MalformedURLException
URL u = new URL(str);
//pode disparar IOException
URLConnection c = u.openConnection();
}
catch (IOException e) {
System.out.println("Não foi possível conectar:" + e);
}
catch (MalformedURLException e) {
System.out.println("Impossível abrir URL:" + e);
}
```

Definir o bloco catch da superclasse primeiro que o da subclasse gerará erro de compilação!

Por fim, vamos ver um exemplo de como pode ser usado o bloco `finally`. Este bloco é útil quando queremos liberar recursos alocados num bloco `try` assim que este terminar.

```

FileInputStream f = null;
try{
f = new FileInputStream(filePath);
while (f.read() != -1)
    charcount++;
}
catch(IOException e) {
System.out.println("Erro ao acessar arquivo" + e);
}
finally {
//Este bloco é sempre executado
f.close();
}

```

O exemplo acima conta quantos caracteres há num arquivo. O arquivo é depois fechado mesmo que a operação de leitura tenha resultado uma exceção.

Um bloco `finally` é executado independente de como o bloco `try` terminou:

- Terminou normalmente, isto é, chegou até o colchete final
- Terminou devido a um `return` ou `break`
- Terminou com o disparo de uma exceção

11.2.2 – Deixando uma exceção passar através do método

Quando uma exceção ocorre mas não temos condição de tratá-la naquele local, podemos passá-la adiante para que num método mais acima ela possa ser tratada adequadamente.

Para liberarmos um método de tratar uma determinada exceção devemos usar o comando *throws* na declaração do mesmo.

```

public int myMethod() throws exception1 {

    //código que pode gerar a exception1

}

```

Exemplo:

```

public URL changeURL(URL oldURL)
    throws MalformedURLException {

return new URL("http://www.xxx.com");

}

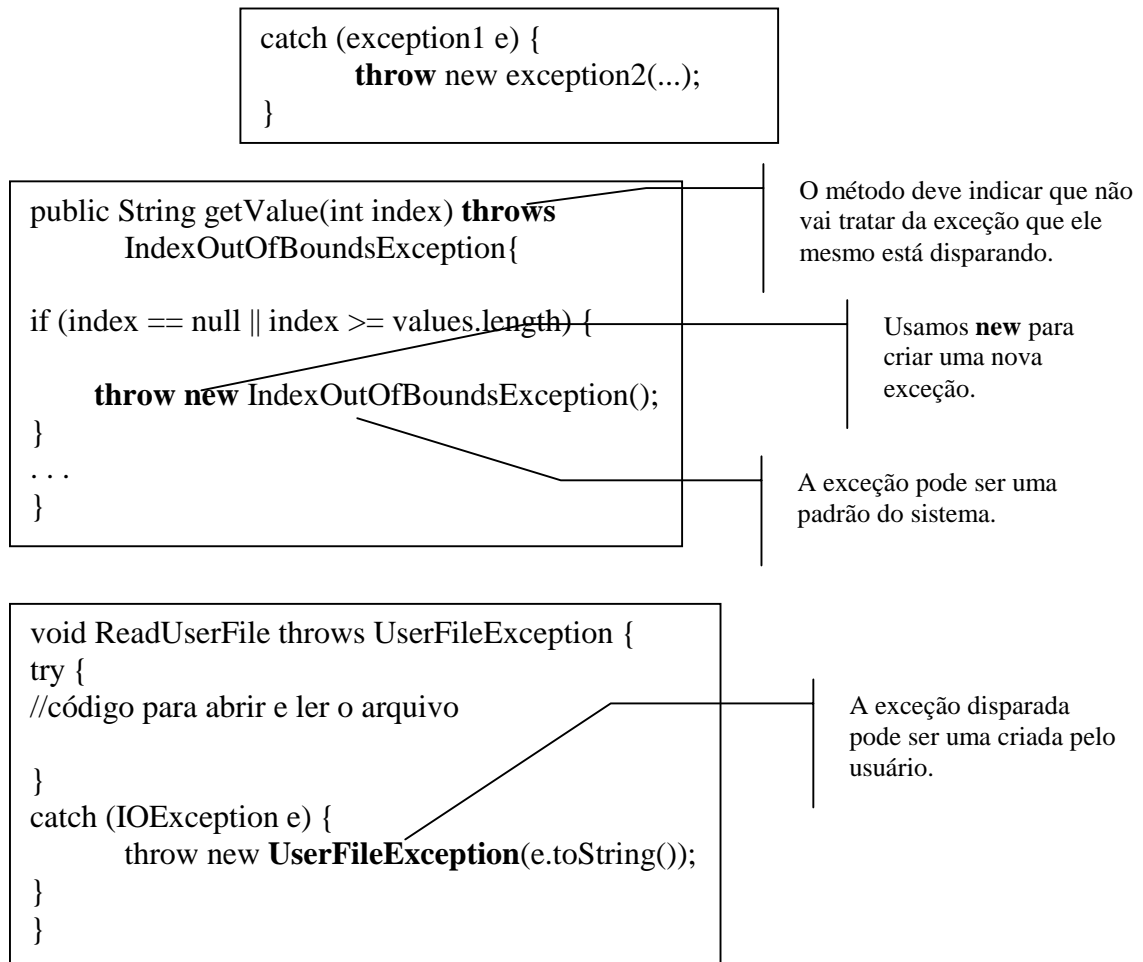
```

Esta linha pode gerar uma exceção de `MalformedURLException`. Porém, o método `changeURL` resolveu não tratá-la e passá-la adiante.

O método que chamou `changeURL` pode optar por tratar a exceção ou também passá-la adiante. Porém, se em nenhum momento algum método tratá-la, o programa terminará em erro.

11.2.3 – Capturando uma exceção e disparando outra diferente

Uma outra alternativa para tratar uma exceção é capturá-la e disparar uma exceção diferente, possivelmente uma criada por nós mesmos.



Exercícios:

Na prática do Capítulo 10 supomos que os métodos `buscaClientePorId` e `buscaItemPorId` sempre recebem IDs válidos para fazer a busca no array de Clientes e Itens respectivamente. Porém, é possível que estes métodos recebam IDs inválidos, isto é, IDs que não estejam cadastrados nos arrays correspondentes.

Iremos fazer a última melhoria em nossa aplicação ACME Vídeo para tratar estes possíveis erros. Abra o projeto relacionado à lição 11.

- 1) Defina no arquivo que contém a classe `ACMEVideo` duas novas exceções, uma para emitir uma mensagem de erro caso um `Cliente` seja inválido e outra quando um `Item` for inválido. As classes derivam de `Exception` e devem se chamar `ClienteNaoEncontrado` e `ItemNaoEncontrado`:

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

- 2) Modifique o método `buscaClientePorId()` para que este emite a exceção apropriada se o `Id` for inválido:
 - a) Adicione o `throws ClienteNaoEncontrado` a especificação do método
 - b) Logo antes do `return`, verifique se o que irá ser retornado é um objeto `Cliente` ou se é `null`. Se for `null`, dispare uma exceção de `ClienteNaoEncontrado`. O código será algo como:

```
if (cliente == null) {
    throw new ClienteNaoEncontrado (“\n\t O ID informado não é válido!” +
        “\n\t Verifique se você entrou com o ID correto.”);
}
```

- 3) Edite agora o construtor da classe `Aluguel`, que é quem faz uso do método `buscaClientePorId`, a fim de capturar a nova exceção. Se o construtor receber uma exceção de `ClienteNaoEncontrado`, o processo de locação não deve continuar.
 - a) Adicione um bloco `try` em torno da chamada do método `buscaClientePorId()`. Adicione um bloco `catch` para tratar da nova exceção:

```
catch (ClienteNaoEncontrado e) {

    System.out.println(e);
    System.exit(0);
}
```

- b) Modifique o exemplo no método `main` que simula uma locação passando um `Id` de cliente inválido, tal como `999`.
- c) Salve, compile e execute sua aplicação

- 4) Altere o método `buscaItemPorId()` para disparar a exceção apropriada se o id recebido for inválido.
 - a) Acrescente o `throw` à definição de `buscaItemPorId()` relacionado à exceção `ItemNaoEncontrado`.
 - b) Antes de retornar o item verifique se este é nulo e, caso afirmativo, crie a nova exceção `ItemNaoEncontrado` tal como foi feito com o cliente.
- 5) Edite agora o construtor da classe `ItemAlugado`, que é quem faz uso do método `buscaItemPorId`, a fim de capturar a nova exceção. Se o construtor receber uma exceção de `ItemNaoEncontrado`, o processo de locação não deve continuar.
 - b) Adicione um bloco `try` em torno da chamada do método `buscaItemPorId()`. Adicione um bloco `catch` para tratar da nova exceção:

```
catch (ItemNaoEncontrado e) {  
  
    System.out.println(e);  
    System.exit(0);  
}
```

- d) Modifique o exemplo no método `main` que simula uma locação passando um Id de item inválido, tal como 999, ao método `addItem`.
 - e) Salve, compile e execute sua aplicação