



# programação

em

# java

Helder da Rocha

Atualizado em 21.09.2015 (Java 8)

# **Conteúdo**

## **Parte I – Java essencial**

1. Introdução
2. Sintaxe
3. Classes e objetos
4. Strings e arrays
5. Coleções
6. Exceções
7. Threads
8. Utilitários
9. Arquivos e I/O
10. Bancos de dados

## **Parte II – Tópicos selecionados**

11. Data e hora (>= Java 8)
12. Concorrência
13. Lambda
14. Streams
15. JavaFX
16. Tratamento de imagens
17. Swing
18. XML e JSON

## **Apêndices**

- A. Exercícios
- B. Referências
- C. Configuração do ambiente: Eclipse, Maven, JUnit e Docker

# Programação em Java

Parte I – Java essencial

# 1 Introdução

---

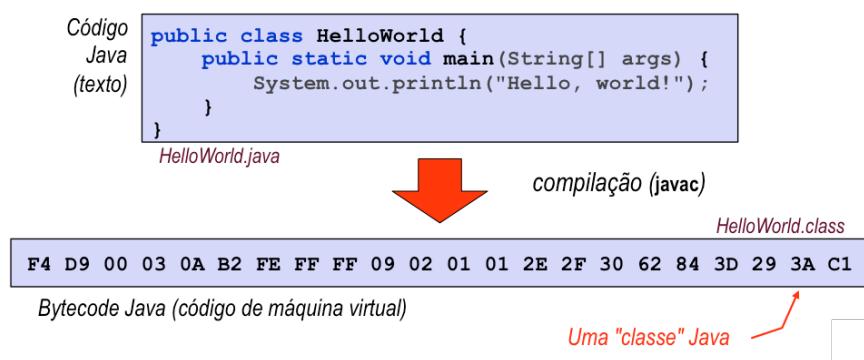
<b>1 O que é um programa em Java</b>	<b>2</b>
<b>2 Ambiente para desenvolvimento de programas em Java</b>	<b>3</b>
<b>2.1 Ant e Maven</b>	<b>3</b>
<b>3 Iniciando em Java com o Eclipse</b>	<b>4</b>
<b>3.1 Criando um projeto</b>	<b>4</b>
<b>3.2 Criando uma classe</b>	<b>5</b>
<b>3.3 Criando uma classe executável</b>	<b>7</b>
3.3.1 Arquivo .class e pacotes	8
3.3.2 Anatomia do método main() e de uma classe executável	9
<b>4 Introdução a classes e objetos</b>	<b>10</b>
<b>4.1 Representando uma abstração com uma classe</b>	<b>10</b>
4.1.1 Usando uma classe como tipo de dados	11
4.1.2 Criação de um objeto usando new	12
4.1.3 Acesso ao estado de um objeto	13
<b>4.2 Métodos e construtores</b>	<b>14</b>
4.2.1 Métodos com parâmetros	15
4.2.2 Encapsulamento, getters e setters	16
4.2.3 A referência this	16
4.2.4 Construtores	17
<b>4.3 Membros estáticos e membros de instância</b>	<b>18</b>
4.3.1 Constantes	19
4.3.2 Mais sobre System.out.println()	19
<b>4.4 Outros componentes de uma classe Java</b>	<b>19</b>
4.4.1 Convenções, espaços e endentação	19
4.4.2 Comentários	20
4.4.3 Extensão	21
4.4.4 Interfaces	21
4.4.5 Enumerações	21
4.4.6 Anotações	21
4.4.7 Classes parametrizadas	22
4.4.8 Instruções	22
<b>5 A Biblioteca Fundamental Java (pacote java.lang)</b>	<b>22</b>
<b>5.1 Object</b>	<b>22</b>
<b>5.2 Class&lt;T&gt;</b>	<b>23</b>
<b>5.3 Number, Character e Boolean</b>	<b>23</b>
<b>5.4 Math</b>	<b>23</b>

<b>5.5</b>	<b>System e Runtime</b>	<b>23</b>
<b>5.6</b>	<b>Process, Thread e Runnable</b>	<b>24</b>
<b>5.7</b>	<b>Throwable, Error, Exception e suas subclasses</b>	<b>24</b>
<b>5.8</b>	<b>Cloneable</b>	<b>24</b>
<b>6</b>	<b>A máquina virtual Java (JVM)</b>	<b>24</b>
<b>6.1</b>	<b>Pilha</b>	<b>25</b>
<b>6.2</b>	<b>Heap e coleta de lixo</b>	<b>25</b>
<b>6.3</b>	<b>Classloader e Classpath</b>	<b>26</b>
<b>6.4</b>	<b>Verificador de bytecode</b>	<b>26</b>

O que é um programa em Java? Como se faz para escrever um programa e executá-lo. Neste capítulo iremos introduzir a linguagem Java de forma prática. Começaremos digitando, compilando e executando um programa simples para configurar o ambiente de desenvolvimento, depois exploraremos algumas outras características de programas em Java para que você seja capaz de analisar um arquivo contendo código-fonte Java e identificar seus componentes. Vários dos tópicos apresentados aqui serão detalhados mais adiante.

## 1 O que é um programa em Java

Um programa em Java é escrito em ou mais arquivos de texto com extensão *.java*. Depois, esses arquivos precisam ser *compilados* - processo que irá gerar arquivos intermediários com extensão *.class*. Um programa em Java é composto por um ou mais arquivos *.class*, e possivelmente outros arquivos auxiliares, chamados de *resources*, organizados em pastas hierárquicas chamadas de *pacotes*. Resources podem ser arquivos de texto com configuração, imagens, arquivos XML, e outros recursos que são usados nos programas. Os arquivos *.class*, resultados da compilação de arquivos *.java*, contém um código de máquina chamado de *bytecode*, que é interpretado por uma *Máquina Virtual Java (JVM)*. A JVM faz parte do ambiente de execução Java (*Java Runtime Environment – JRE*), que provavelmente está instalado no seu sistema operacional.



Programas em Java são geralmente distribuídos em estruturas de diretórios (pacotes) empacotados em arquivos com compressão ZIP, e com extensão *.jar*, *.war*, *.ear*, etc. dependendo da sua finalidade.

Um programa em Java pode ser construído para executar como aplicação *standalone* do sistema operacional, através da JVM. Para isto é necessário que uma de suas classes seja configurada como executável através da presença de uma operação de *bootstrap*, que dá partida na execução:

o método *main*. Nem todo programa em Java tem essa finalidade. Muitos são criados para serem *bibliotecas* ou *frameworks*, usados por outros programas. Outros são *componentes* que precisam ser implantados em um *container* que irá fornecer seu ambiente de execução.

## 2 Ambiente para desenvolvimento de programas em Java

Para escrever e executar um programa em Java é necessário ter:

- Um *editor de texto*, onde o código possa ser digitado e gravado com a extensão *.java*,
- Um compilador, para processar o arquivo *.java* e convertê-lo em um ou mais arquivos *.class*, e
- Um ambiente de execução Java, ou *JRE*, para executar o programa.

A maioria dos sistemas operacionais possuem JREs nativos (mas nem todas as configurações permitem o uso em uma janela de terminal). Você pode verificar se na sua plataforma existe um JRE instalado e acessível, abrindo uma janela de terminal e digitando:

```
java -version
```

que irá rodar o JRE e exibir a versão instalada.

Poucas plataformas, porém, têm um *ambiente de desenvolvimento Java* instalado (o *JDK* ou *Java SDK*). Ele pode ser instalado baixando o pacote JDK da Oracle, e é útil para aprender Java e experimentar com alguns programas simples, mas ele consiste apenas de ferramentas de linha de comando. Você precisa criar seu próprio ambiente de desenvolvimento integrando um editor de textos, cuidando da organização dos projetos em pastas, e executando os programas através de janelas do terminal.

É muito mais fácil desenvolver em Java usando um *ambiente integrado de desenvolvimento*, ou *IDE*. Os mais populares, o *Eclipse* e *NetBeans*, são de graça e podem ser instalados em qualquer plataforma. Eles oferecem um ambiente adequado a projetos maiores, organizando os arquivos em pastas, configurando dependências, facilitando execução e depuração, colorindo e formatando automaticamente o código, e disponibilizando documentação contextual. Elas também cuidam do processo de construção da aplicação. Esse processo pode ser bem simples, se o programa for simples (talvez seja apenas uma etapa de compilação),, mas em outros casos pode ser necessário baixar dependências, compilar, empacotar, testar, implantar e instalar em um servidor, o que justifica usar uma ferramenta como uma IDE para administrar esse processo.

### 2.1 Ant e Maven

Um problema comum das IDEs é que a forma como organizam e constroem os projetos não é padronizada. Isto dificulta o compartilhamento de código em projetos maiores, pois requer que suas dependências, estrutura e outras configurações tenham que ser refeitas caso haja necessidade de trabalhar com a aplicação em uma IDE diferente daquela que foi usada para construir a aplicação, ou mesmo para integrá-la a um ambiente de integração contínua, necessidade comum em projetos que são desenvolvidos em processos ágeis, DevOps ou Scrum.

A solução nesses casos é ter o processo de construção separada da IDE. Ferramentas como *Ant*, *Ivy* e *Maven* incluem a estrutura e processo de construção de uma aplicação como parte do projeto, configurável através de arquivos XML que podem ser versionados, permitindo que sejam montadas em diferentes IDEs e em processos automatizados. Mas não é preciso abandonar a IDE. O NetBeans usa Maven como método nativo de construção e gerenciamento de um projeto Java, e oferece suporte a Ant/Ivy via plug-ins. O Eclipse também suporta Maven e Ant, via plug-ins.

A maior parte dos exemplos usados neste tutorial estão em projetos configurados com o Maven, e podem ser usados em qualquer IDE. Para usar no Eclipse, é preciso instalar o plug-in M2E através do Eclipse Workspace, e importar os exemplos para a workspace.

## 3 Iniciando em Java com o Eclipse

Neste tutorial inicial usaremos o *Eclipse* para desenvolver aplicações em Java. Vamos começar com uma aplicação simples e executável em linha de comando.

Você deve ter o Eclipse instalado no seu ambiente. Se não tiver, você pode baixa-lo e instalá-lo a partir de:

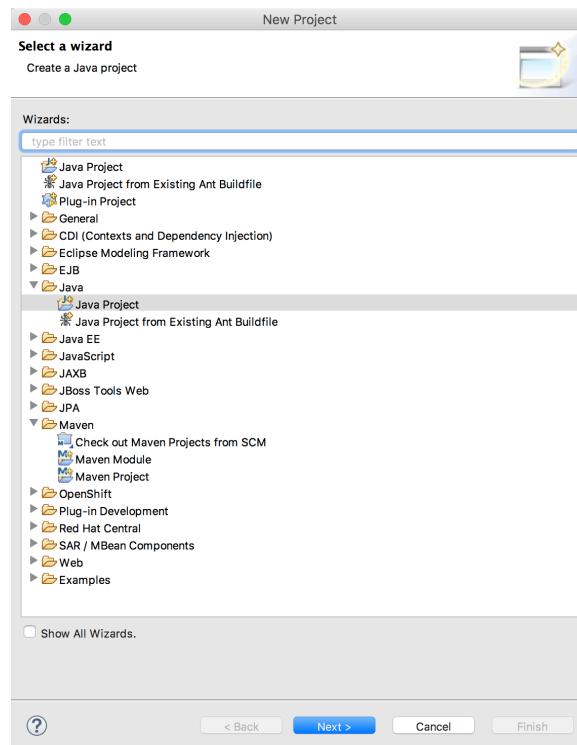
<https://eclipse.org/downloads/>

Neste tutorial usamos a distribuição Eclipse Neon, portanto as telas poderão ser um pouco diferentes se você estiver usando alguma outra versão.

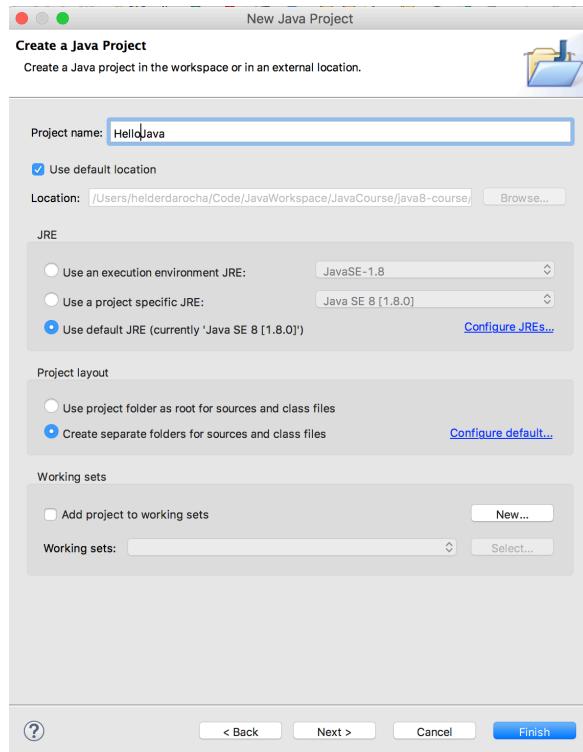
### 3.1 Criando um projeto

O primeiro passo é criar um novo *projeto*. Um projeto é uma pasta dentro da qual são guardadas todas as informações necessárias para construir uma aplicação. Deve conter uma pasta para código-fonte, e provavelmente terá pastas para código compilado, dependências e pacote de distribuição, que poderão ser geradas durante o desenvolvimento.

Para criar um projeto Java simples, abra o menu *File/New/Project...* No assistente de *New Project* (tela abaixo) selecione *Java/Java Project*.

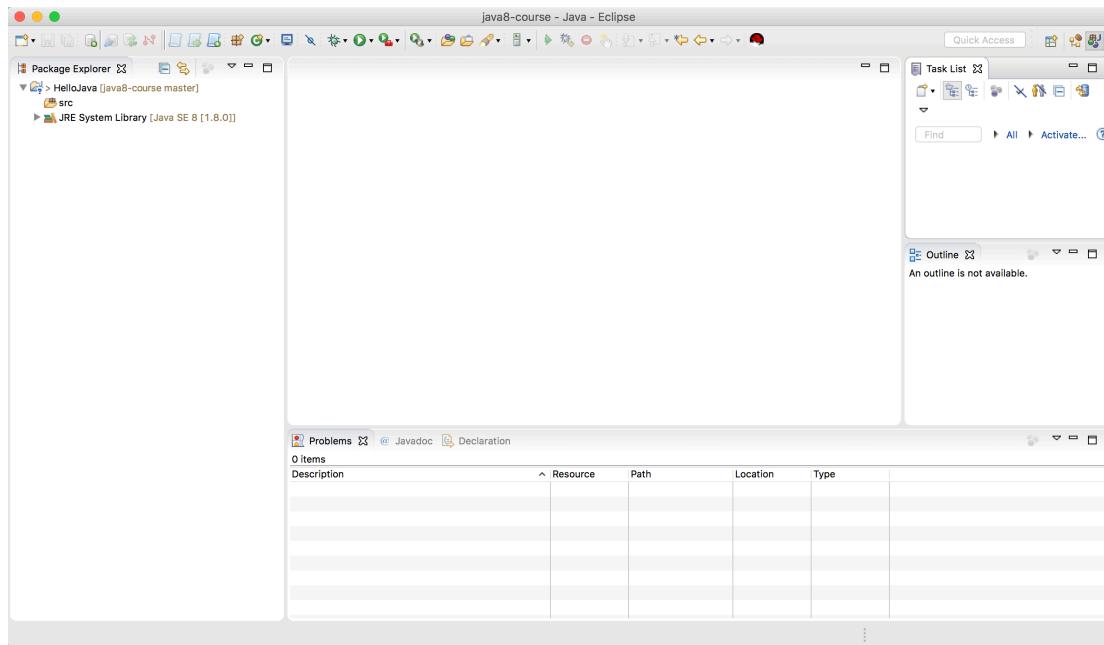


Clique *Next*. Na tela seguinte, digite “HelloJava” no campo *Project Name*:



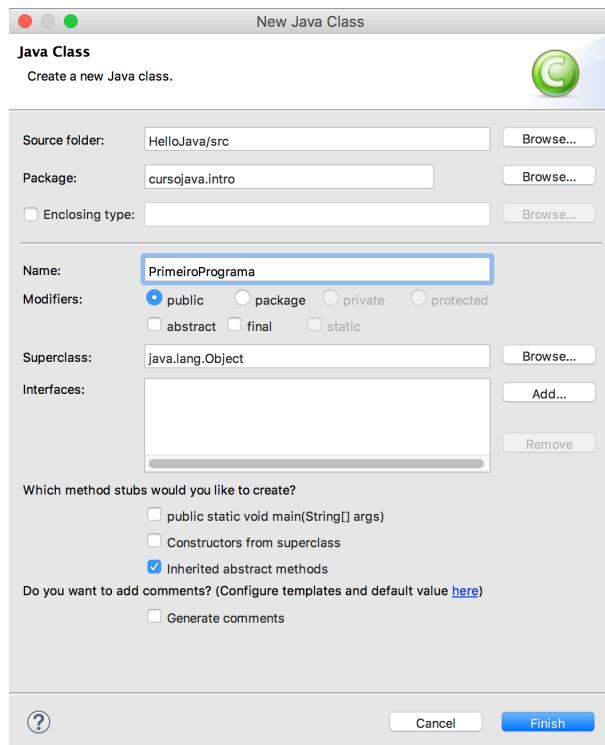
Verifique que o JRE usado é Java SE 8 (se não for o *default*, veja se é possível selecioná-lo na lista.) Clique *Next*. Nesta janela o Eclipse informa que nosso código-fonte será armazenado em uma pasta *src* (dentro do projeto). Isto é o padrão no Eclipse. Na última caixa de entrada o Eclipse informa onde o código compilado (arquivos *.class*) serão colocados (*default* no Eclipse é *HelloJava/bin*). Aperte *Finish*.

A sua área de trabalho deverá ser similar a esta:



## 3.2 Criando uma classe

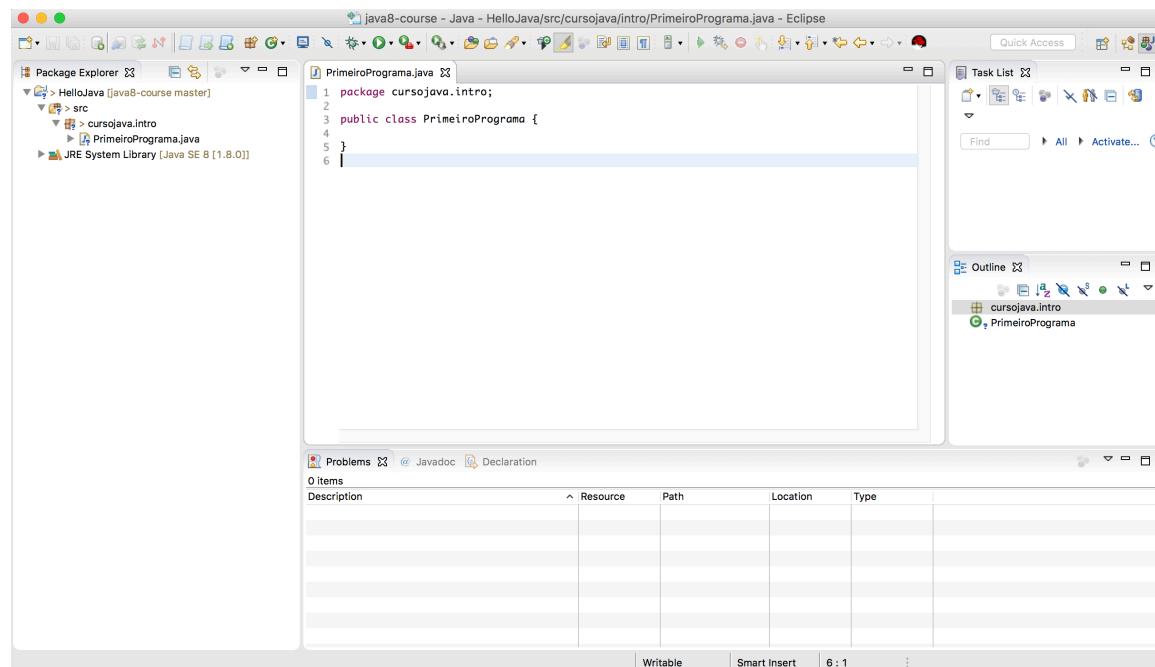
Clique agora no projeto *HelloJava*, e selecione no menu a opção *File/New/Class*. Vamos criar uma classe Java, é que a base para qualquer aplicação. Preencha os dados de acordo com a tela abaixo:



Digite “*cursojava.intro*” no campo *Package*. Apesar de não ser obrigatório, declarar um pacote é uma prática fortemente recomendada. Ele cria um *namespace* que protege a classe de conflitos de nomes, algo que pode acontecer mais facilmente em projetos maiores.

Digite *PrimeiroPrograma* no campo *Name*. Este é o nome da classe e também será o nome do arquivo *.java*. Lembre-se de escrever o nome exatamente como foi sugerido, respeitando as maiúsculas e minúsculas. Java é uma linguagem case-sensitive, e isto se aplica aos nomes dos arquivos também. Usar caixa-mista (e não apenas maiúsculas e minúsculas) é uma convenção do Java e uma boa prática. Digite *Finish*.

A tela do Eclipse agora mostra o programa na área principal, e no Package Explorer, o arquivo *PrimeiroProgramma.java* aparece dentro do pacote *cursojava.intro*, dentro da pasta *src*.



Há muitos recursos no Eclipse que exigiriam outro tutorial para explicar, e é mais proveitoso fazer isto quando você já tiver mais prática em programação. Depois de algumas semanas programando em Java, tome um tempo e leia um tutorial sobre o Eclipse, para descobrir seus recursos.

Observe que a declaração de pacote aparece no início do arquivo PrimeiroPrograma.java, e termina em ponto e vírgula. O nome *package* é uma palavra reservada, assim como *public* e *class*. Elas têm significado especial e o editor do Eclipse põe as duas em destaque. Observe que a declaração *public class* não termina em ponto-e-vírgula mas em um bloco delimitado por chaves.

Toda a definição da classe e todas as operações do nosso programa acontecem *dentro* dessas chaves. As únicas instruções que podem vir fora de uma declaração de classe são outras declarações de classe (ou de interface ou enum, que são parecidas), uma instrução *package* (antes de tudo) e instruções *import* (depois de *package*), além de comentários.

O Eclipse, por default, compila o código automaticamente. Portanto o código mostrado já foi compilado e não contém erros.

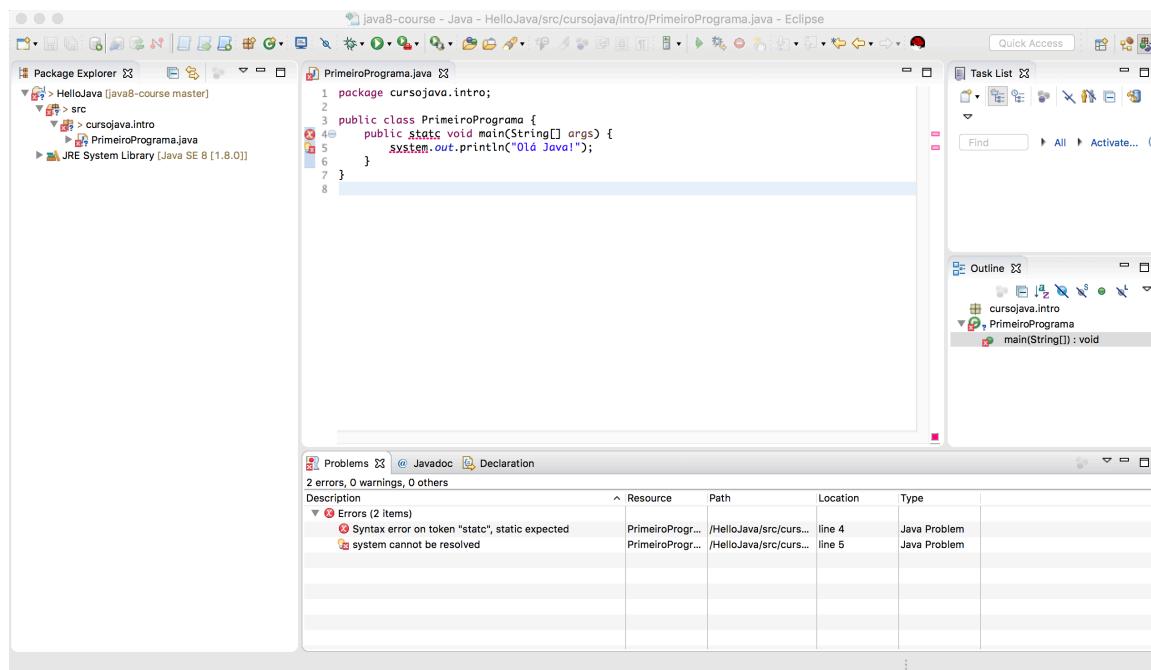
### 3.3 Criando uma classe executável

Vamos digitar um *método* dentro da classe *PrimeiroProgramma*. Um método é uma *operação*. É similar (mas não igual) ao que em outras linguagens se chama de função, procedimento ou subrotina. Digite o código a seguir, destacado em negrito, dentro das chaves, exatamente como está escrito, respeitando os formatos maiúsculo e minúsculo do texto:

```
package cursojava.intro;

public class PrimeiroProgramma {
    public static void main(String[] args) {
        System.out.println("Olá Java!");
    }
}
```

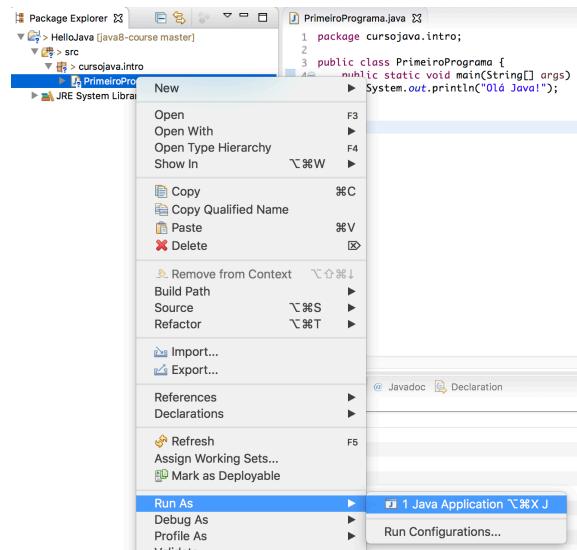
Se houver erros, o Eclipse mostrará imediatamente. Se você *gravar* o arquivo (use o menu ou Ctrl/Cmd-S) a lista de erros irá aparecer na aba *Problems*, na parte inferior do Eclipse:



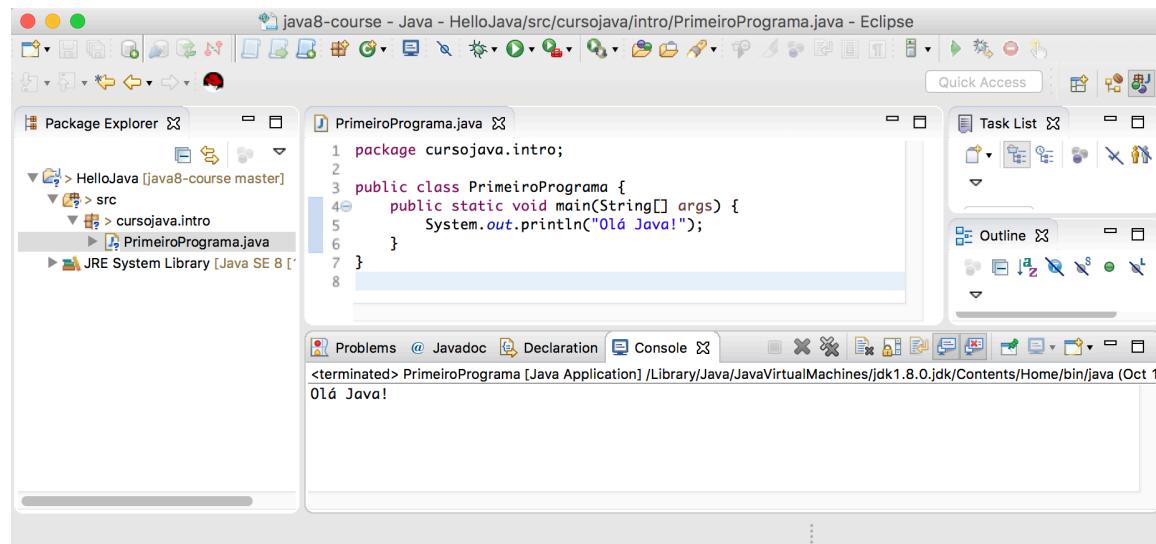
Neste caso, a palavra *static* foi escrita incorretamente (é um erro de sintaxe) e a palavra *System*, que *não é* uma palavra reservada (observe que ela não aparece em outra cor), causou um erro diferente. Ela não pôde ser *resolvida*. *System* é o nome de uma outra *classe* que está sendo usada por nosso programa. Mas neste exemplo escrevemos *system*, com “s” minúsculo, e por isso o Java não a encontrou. Java é case-sensitive, portanto faz diferença usar maiúsculo ou minúsculo.

Se houver erros no seu programa, corrija-os. Grave o arquivo (não devem aparecer mais problemas na aba *Problems*).

O programa que criamos é *executável*. Para rodá-lo clique o botão direito sobre o nome do arquivo no *Package Explorer*, depois selecione a opção *Run As* no menu e depois a opção *Java Application* no submenu.



O programa irá executar e seu resultado será mostrado numa nova Janela, o *Console*:



### 3.3.1 Arquivo .class e pacotes

Nós executamos o programa pelo Eclipse, mas também podemos executar o programa usando o programa *java*, da JRE, em linha de comando. Descubra a pasta onde o Eclipse gravou o seu projeto e mude até o diretório onde está instalado o projeto *HelloJava*. Entre nele, e depois entre no diretório *HelloJava/bin* e veja seu conteúdo:

```
cursojava/
intro/
PrimeiroProgramma.class
```

Abra uma janela de *terminal* do seu sistema operacional dentro da pasta *HelloJava/bin*. Se seu ambiente permitir executar o JRE em linha de comando, digite:

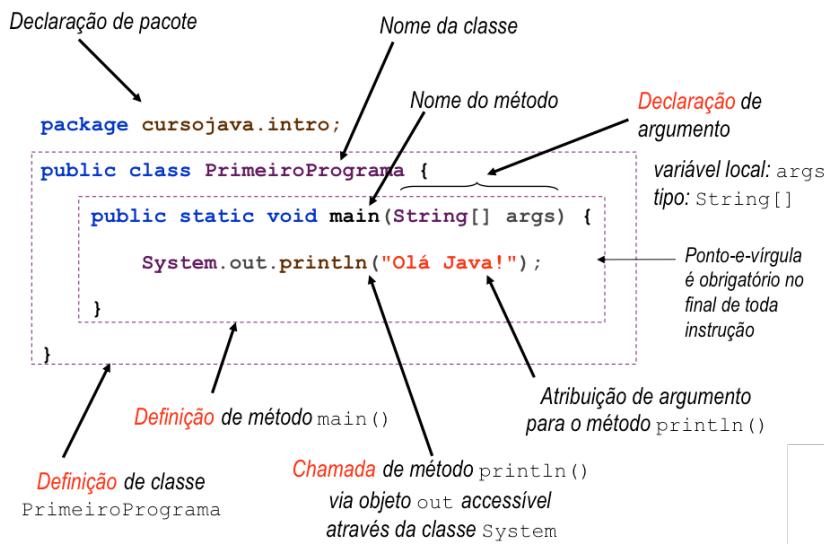
```
java cursojava.intro.PrimeiroProgramma
```

E o programa irá executar da mesma forma que no Eclipse.

### 3.3.2 Anatomia do método main() e de uma classe executável

O arquivo que criamos é uma *classe Java*. Uma *classe*, no paradigma orientado a objetos, é uma *abstração* usada para *representar* alguma coisa concreta, uma idéia, um conceito. Uma classe também é usada como *molde* para criar estruturas de dados que chamamos de *objetos*. Quando modelamos uma aplicação Java e projetamos suas classes, geralmente pensamos nos *dados* representados por essas classes (seu estado) e a *interface* (comportamento) que ela define.

Abaixo uma descrição da estrutura da classe (não se preocupe se você não entender todos os componentes descritos):

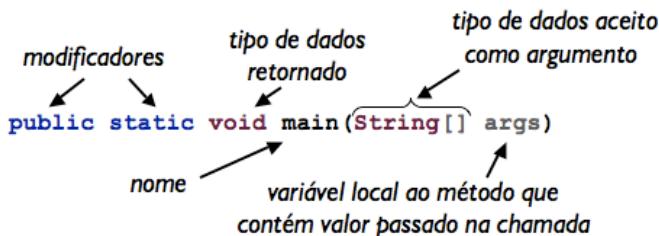


A classe que criamos neste exemplo é especial, pois ela não mantém um estado (ela simplesmente executa uma operação), e a interface que ele define (o método *main()*) é a interface padrão para *programas executáveis*. O programa *java* conhece essa interface, por isso sabe como executar. O método *main* tem a seguinte *assinatura*:

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

O programa *java* espera que qualquer classe passada para ele tenha um método *main()* escrito exatamente da maneira acima. Se você passar uma outra classe Java que não tenha método *main* escrito exatamente dessa maneira, ele irá apresentar uma mensagem de erro.

O desenho abaixo resume a anatomia do método *main()*:



- O modificador *public* indica que o método faz parte da *interface pública* da classe, e pode ser chamado de fora dela. Isto é necessário para que o programa *java* possa executá-la.
- O modificador *static* garante que o método *main()* estará disponível tão logo a classe for carregada, sem a necessidade de criar objetos. Isto é importante porque o programa *java* não cria objetos com *PrimeiroPrograma.class* (depois mostraremos como fazer isto), mas apenas tenta executar o *main()*, que deverá estar disponível.
- A palavra *void* indica o tipo do valor retornado pelo *main()* depois que ele executa. Muitos métodos, executam uma lista de operações que produzem um valor, que é retornado. Como este método apenas imprime na saída e não retorna nada, usa-se a palavra *void*.
- Finalmente, o método recebe um *parâmetro* ou *argumento* que tem um tipo de dados declarado: *String[]*, que significa *array de String*. Quando o programa *java* executar a classe, ele irá coletar quaisquer argumentos de linha de comando, guardar cada palavra em um *array*, e passar esse array como argumento para o *main()* durante a execução. Se não houver argumentos (como no nosso caso), o programa *java* passará para o *main()* um array de *Strings* vazio.

Dentro do *main()* temos apenas *uma* instrução:

```
System.out.println("Olá Java");
```

Que é usada para imprimir texto na saída padrão de um terminal (console). O valor entre aspas duplas é uma *literal* do tipo *String*. Qualquer tipo em Java pode ser convertido em *String*, e *System.out.println()* serve para imprimir a representação *String* de qualquer coisa.

## 4 Introdução a classes e objetos

Podemos escrever longos programas em Java usando apenas uma classe e colocando todas as instruções dentro do método *main()*. Isto seria programar em Java de *modo procedural*, e subutilizar os recursos da linguagem. Como Java é uma linguagem *orientada a objetos*, para aproveitar bem seus benefícios a aplicação deve ser projetada de forma que a solução do problema seja expressa de forma orientada a objetos, priorizando os dados (objetos) em vez das instruções (procedimentos), e fazendo uso eficiente de abstração, polimorfismo, encapsulamento e outras características fundamentais do paradigma orientado a objetos.

### 4.1 Representando uma abstração com uma classe

Precisamos criar uma aplicação que irá representar diferentes pontos sobre um sistema de coordenadas, para desenhá-los numa tela em pixels. Cada ponto tem uma coordenada *x* e *y*, que recebem um número *inteiro*. Inicialmente precisamos criar um programa que represente 3 pontos na tela. Podemos começar criando uma classe.

Vamos coloca-la em seu próprio pacote: *cursojava.intro.geometria*. Use o Eclipse e crie mais uma classe, a acrescente a linha destacada abaixo:

```
package cursojava.intro.geometria;

public class Ponto {
    public int x, y;
}
```

Esta linha *declara* duas *variáveis*, *x* e *y*. Estas variáveis representam o estado do objeto, e são chamadas de *campos de dados*, *atributos*, ou *variáveis de instância*. A palavra *int* é usada para identificar números inteiros de propósito geral. Portanto, *int* declara que essas variáveis poderão receber valores do tipo *int*, quando um objeto da classe *Ponto* for usado.

O fato dessa declaração ocorrer *dentro da declaração da classe*, define as variáveis como variáveis de instância, que representam o estado dos objetos que serão criados com esta classe. Variáveis declaradas em outros lugares (como, por exemplo, dentro de métodos, têm uma função diferente). Elas são declaradas com o modificador de acesso *public* para que possam ser lidas e alteradas por outras classes.

#### 4.1.1 Usando uma classe como tipo de dados

Uma classe representa um *tipo de dados*. Jává possui vários tipos que chamamos de primitivos, pois eles estão embutidos na linguagem. Exemplos são *int*, *double*, *char*, *byte*. Esses tipos são usados para representar dados escalares, *unidimensionais*.

Para representar tipos mais complexos é preciso criar *classes*. Por exemplo, uma abstração que representa uma *Data* (com *dia*, *mes*, *ano*) poderia ser criada da forma:

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

Como os três atributos são do mesmo tipo, também é possível declarar o tipo apenas uma vez:

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

Depois, esse tipo novo pode ser usado para declarar o tipo de variáveis que representam datas, por exemplo:

```
Data hoje;
```

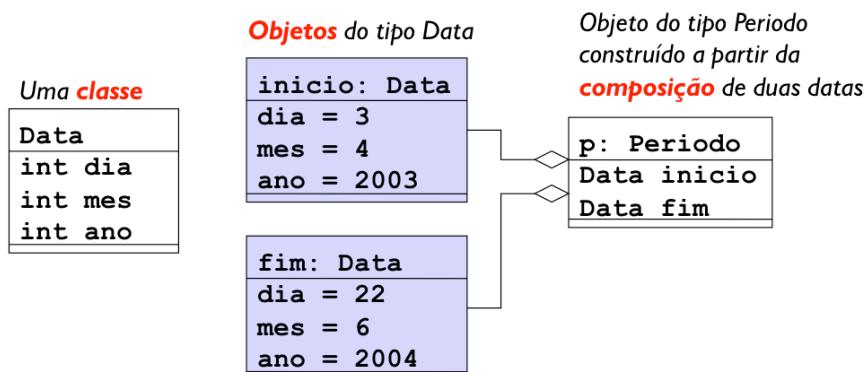
Seus atributos, se publicamente acessíveis podem ser recuperados com o operador ponto “.”:

```
int diaDeHoje = hoje.dia;  
int mesDeHoje = hoje.mes;  
int anoDeHoje = hoje.ano;
```

E essa classe poderia ser usada para criar tipos ainda mais complexos, que são *compostos* por datas. Por exemplo, um tipo *Periodo* poderia ser construído com duas datas:

```
class Periodo {  
    Data inicio, fim;  
}
```

O desenho abaixo representa, em UML, a composição de duas datas para formar um *Periodo*:



Portanto, também podemos usar *Ponto* para declarar o tipo de uma variável, da mesma forma como declararmos o *x* e *y* em cada Ponto.

Para usar a classe *Ponto* que acabamos de criar, crie um novo programa executável *MapaDePontos* no pacote *cursojava.intro*:

```

package cursojava.intro;

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

```

Para usar a classe *Ponto*, precisamos importá-la usando uma instrução *import*. Essa instrução precisa indicar o *nome completo* da classe que vamos importar, e deve estar em uma linha entre a declaração *package* e a declaração de classe. O nome completo é o nome do pacote, concatenado com o nome da classe através de um ponto:

```
import cursojava.intro.geometria.Ponto;
```

Vamos declarar, dentro do método *main()*, três variáveis locais do tipo *Ponto*:

```
Ponto p1, p2, p3;
```

As variáveis *p1*, *p2* e *p3* representam objetos que ainda não foram criados, e irão armazenar *referências* para eles.

As variáveis *p1*, *p2* e *p3* são chamadas de *variáveis locais* porque foram declaradas *dentro do método*. Compare com as variáveis de instância *x* e *y* que foram criadas *dentro da classe Ponto*. Variáveis locais têm *escopo local*, ou seja, deixam de existir quando o método termina (por isso não precisam nem podem ter modificadores *public* ou *private*, como as variáveis de instância).

#### 4.1.2 Criação de um objeto usando new

Para criar um objeto é preciso alocar espaço para ele na memória. Esta alocação ocorre através da operação *new* seguido do *construtor* do objeto. O construtor é parte de uma operação especial criada automaticamente pelo sistema e que *inicializa* as variáveis de instância a valores *default* (no caso do *Ponto*, *x* e *y* terão valores 0, que é o *default* para *ints*).

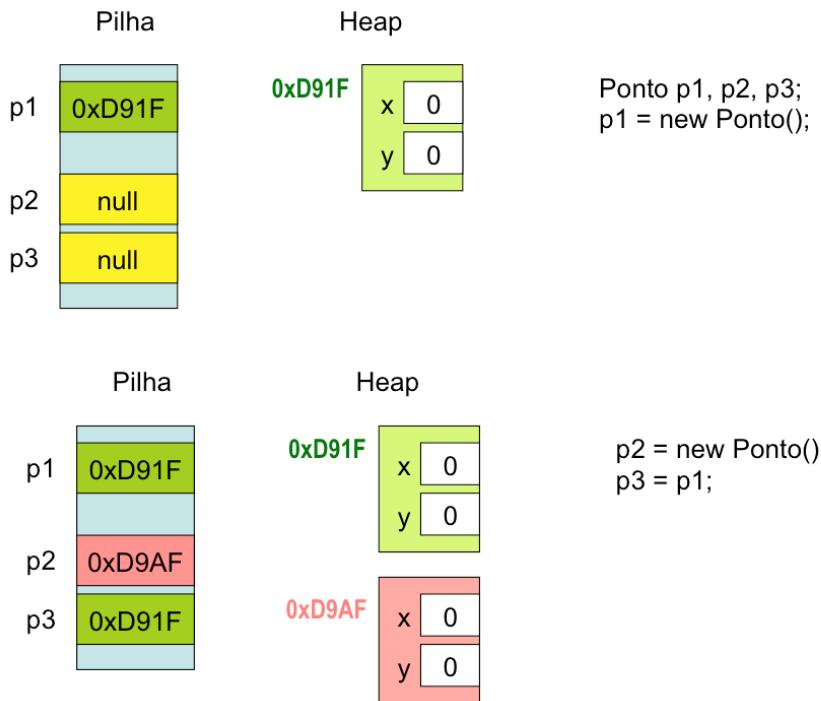
Uma operação *new* aloca espaço numa área de memória que permite alocação aleatória (o *heap*), realiza uma série de operações para inicializar o objeto (entre elas atribuir valores iniciais a seus atributos) e retorna uma *referência* para o objeto. Uma referência é uma ligação de acesso ao objeto. Também é chamado de *ponteiro* - o *endereço na memória* onde o objeto está armazenado.

Todo acesso ao objeto é realizado através da referência. Um objeto pode ter várias referências. Se em algum momento ele perder todas suas referências, ele não poderá mais ser acessado e o sistema irá marcá-lo para reciclagem.

Para usar o objeto não é suficiente chamar seu construtor com *new*. É preciso guardar a referência em algum lugar para que seja possível ter acesso aos seus dados e chamar seus métodos. Uma das formas de guardar a referência para um objeto é copiá-la para uma variável através de uma *operação de atribuição*, usando o operador *=*. Abaixo atribuímos a referência de um objeto *Ponto* à variável *p1*. Isto só é possível porque *p1* foi declarada como sendo uma variável *do tipo Ponto*, e portanto aceita receber como valor uma referência para objetos *Ponto*:

```
p1 = new Ponto();
```

O desenho abaixo ilustra, de forma esquemática, o que ocorre quando um objeto *Ponto* é instanciado e sua *referência* é *copiada* para uma variável. Observe que a cópia de uma referência (realizada através de uma operação de atribuição com o operador *=*) *não cria um objeto novo* (*não aloca memória para um objeto*). Apenas a instrução *new* efetivamente cria um objeto, alocando memória para ele no *heap*. A variável na pilha contém apenas o endereço (referência) deste objeto:



Em Java *nunca* é necessário remover objetos da pilha (ou liberar memória no heap). Isto é feito automaticamente através de um algoritmo de coleta de lixo que executa de forma automática na JVM. Pode-se marcar objetos para remoção eliminando todas as suas referências (declarando-as com *null* ou copiando outras referências para elas). A memória da pilha é sempre liberada automaticamente quando um método termina.

#### 4.1.3 Acesso ao estado de um objeto

Uma vez criado o objeto, memória é alocada para suas variáveis de instância, e valores iniciais são armazenados. Para acessar os *membros públicos* de um objeto, usamos o operador ponto “.”. Com ele poderemos ler os valores das variáveis *x* e *y* do objeto referenciado por *p1* usando as expressões *p1.x* e *p1.y*.

Para imprimir o valor de *p1.x*, por exemplo, podemos usar uma expressão *System.out.println* da forma:

```
System.out.println("Valor de p1.x = " + p1.x);
```

O “+”, quando usado em uma expressão que contém Strings, funciona como *concatenação*. O valor que está fora das aspas é um *int*, mas ele será transformado em *String* durante a concatenação. A seguir a listagem do *main()* para imprimir os dois valores:

```
public static void main(String[] args) {
    Ponto p1, p2, p3;
    p1 = new Ponto();
    p2 = new Ponto();
    p3 = new Ponto();

    System.out.println("p1.x: " + p1.x);
    System.out.println("p1.y: " + p1.y);
}
```

Execute a aplicação *MapaDePontos* contendo o *main()* acima. O resultado deve ser:

```
p1.x: 0
p1.y: 0
```

Podemos fazer o mesmo para imprimir os valores de *p2* e *p3*. Os resultados serão iguais, porque os valores de *x* e *y* foram inicializados com defaults, mas os pontos são diferentes e ocupam lugares distintos na memória.

Altere o valor de algumas variáveis atribuindo inteiros diferentes através de operações de atribuição, antes de chamar *System.out.println()*:

```
p1.x = 50;
p2.x = 10;
p1.y = 125;
```

Agora imprima todos os valores de *p1*, *p2* e *p3* e vejam que eles são diferentes.

## 4.2 Métodos e construtores

Um objeto pode encapsular *estado* (seus dados) e também *comportamento* (operações relacionadas a seus dados). Podemos acrescentar um *método* na classe *Ponto* que calcula a distância do ponto até a origem do sistema de coordenadas.

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

    public double distancia() {
        double d = Math.sqrt(x * x + y * y);
        return d;
    }
}
```

*Math.sqrt()* é outra função nativa do Java, como o *System.out.println()*. Ela calcula a raiz quadrada e retorna o resultado como um número do tipo *double* (ponto flutuante de dupla precisão). Na primeira linha do método *distancia()* calculamos o quadrado dos valores de *x* e *y*, que foi passado como argumento para o método *Math.sqrt()*. O resultado, *retornado* pelo método *sqrt()* foi atribuindo à variável *d*. Esta variável tem que ser declarada como *double*, porque esse é o tipo que o método *sqrt()* retorna.

Como sabemos que o valor retornado é do tipo *double*? Lendo a documentação da classe *Math*, mas o Eclipse pode nos ajudar com isto também!

Na última linha temos a instrução *return d*, que devolve esse valor à instrução que chamar o método. Observe a declaração do método:

```
public double distancia()
```

Esta declaração é chamada de *assinatura* do método. A assinatura mínima de um método em Java tem o formato:

```
tipo-de-retorno nome()
```

Opcionalmente a declaração de um método pode ser precedida por uma lista de *modificadores* (ex: *public*, *private*, *static*, *final*, *abstract*, *synchronized*, etc.). Pode também conter uma lista de *parâmetros* separados por vírgula entre parênteses, e uma declaração de exceções. Por exemplo:

```
public int registrar(String nome, int idade, double mensalidade)
                    throws IOException { ... }
```

O método *distancia()* possui apenas um modificador: *public*. Isto permite que este método seja chamado de fora da classe. Ele precisa declarar *double* como tipo de retorno, porque este é o tipo da variável *d*.

Podemos agora calcular a distância do ponto *p1* à origem do sistema de coordenadas, e imprimir o resultado:

```
System.out.println("Distancia de p1 à origem: " + p1.distancia());
```

Executar o programa *MapaDePontos* e verificar o resultado:

```
Distancia de p1 à origem: 134.6291201783626
```

Observe que o método *distancia()* não precisou passar parâmetros porque ele operou sobre os dados do próprio objeto. Essa é uma característica dos métodos de instância. Eles *fazem parte do objeto*. Eles não fazem parte da classe. A classe em si não atribui valores para x e y. Ela é apenas uma meta-representação do objeto. Se você executar *p2.distancia()* o resultado será outro, pois o método que está dentro do objeto vai encontrar outros valores de x e y, os que estão no objeto referenciado por p2.

#### 4.2.1 Métodos com parâmetros

Para calcular a distância entre dois pontos, poderíamos utilizar várias estratégias. Uma delas seria escrever um método *dentro da própria classe Ponto*, que calcule a distância do *ponto atual* com um segundo ponto recebido como parâmetro do método:

```
public double distancia(Ponto p) {
    long q = (x - p.x) * (x - p.x) + (y - p.y) * (y - p.y);
    return Math.sqrt(q);
}
```

O método espera receber um *parâmetro do tipo Ponto*, portanto, para chamar o método é preciso passar outro ponto como argumento. Para calcular a distância entre p1 e p2 pode-se usar:

```
System.out.println("Distancia entre p1 e p2: " + p1.distancia(p2));
```

ou:

```
System.out.println("Distancia entre p1 e p2: " + p2.distancia(p1));
```

O primeiro chama um método em *p1*, e passa *p2* como argumento. O segundo chama um método em *p2*, e passa *p1* como argumento. Neste caso, ambos produzem o mesmo resultado.

Os dois métodos com o mesmo nome *distancia()* podem existir na mesma classe, desde que tenham *quantidades diferentes de parâmetros*, ou *parâmetros de tipos diferentes*. Isto é chamado de *sobrecarga de nomes*, e é muito comum em Java.

Uma outra estratégia é criar um outro método na classe *MapaDePontos* que receba dois pontos. Este método foi declarado *static*, para pode ser chamado diretamente de dentro do *main()* sem precisar instanciar um objeto (mais sobre isto adiante):

```
public class MapaDePontos {
    public static void main(String[] args) { ... }
    public static double distancia(Ponto a, Ponto b) {
        long q = (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
        return Math.sqrt(q);
    }
}
```

Observe que, como o método compara os valores x e y do *Ponto a*, com os valores x e y do *Ponto b*, essas variáveis foram sempre *prefixadas* com *a.* ou *b..* Para chamar este método de dentro do *main()* declarado na mesma classe, é preciso passar *dois* pontos, mas o método em si não está associado a nenhum objeto ou classe:

```
System.out.println("Distancia entre p1 e p2: " + distancia(p1, p2));
```

A distância também poderia ser calculada *sem argumentos*. Se no nosso projeto houver uma classe *Linha*, que é definida por dois pontos, o *comprimento* da linha seria essa distância. Como a *Linha* contém todos os pontos envolvidos, o método não precisa de parâmetros:

```
package cursojava.intro.geometria;

public class Linha {
    public Ponto a, b;
    public double comprimento() {
```

```

        long q = (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
    return Math.sqrt(q);
}

```

E poderia ser chamado simplesmente usando:

```
System.out.println("Comprimento da linha: " + linha.comprimento());
```

#### 4.2.2 Encapsulamento, getters e setters

*Encapsulamento* é uma das características de linguagens orientadas a objeto. Consiste em esconder o que não é essencial para usar um objeto. A *interface pública* de um objeto determina as partes que podem e as que não podem ser acessadas por outras classes. Os métodos garantem um nível de encapsulamento ao não permitir que suas variáveis locais sejam acessadas fora deles. Mas para limitar o uso de variáveis de instância é preciso usar *modificadores de acesso*.

Há três modificadores de acesso em Java:

- *public* – permite acesso a qualquer classe dentro ou fora do seu pacote
- *protected* – permite acesso a subclasses e classes dentro do pacote
- *private* – só permite acesso dentro da própria classe

Não usar modificador nenhum limita o acesso apenas a *outras classes dentro do mesmo pacote* (ex: a classe *Linha* poderia acessar as variáveis de *Ponto* declaradas dessa forma, pois estão no mesmo pacote).

Restringir o acesso a variáveis de instância é uma boa prática. Com variáveis públicas não é possível controlar nem limitar os valores que são atribuídos. Também não é possível estabelecer que uma variável será somente leitura. Por exemplo, não há como impedir que coordenadas x ou y negativas sejam definidas, mesmo que isto não seja permitido de acordo com os requisitos da aplicação.

A prática comum e recomendada é *encapsular* as variáveis, declarando-as como *private*, e prover *métodos de acesso e mutação*, chamados de *getters* e *setters*, com a sintaxe:

```

public tipo getAtributo() {
    return atributo;
}
public void setAtributo(tipo novoAtributo) {
    atributo = novoAtributo;
}

```

O primeiro método simplesmente *retorna* o valor de *atributo*. O segundo recebe um valor novo e o usa para *alterar* o valor de *atributo*.

Com os atributos *x* e *y* declarados como *private*, não é mais possível chama-los diretamente, toda operação que precisar ler ou gravar *x* ou *y* em uma classe externa precisa usar os métodos *getX()*/*getY()* e *setX()*/*setY()* em vez de *x* e *y*. Em vez de:

```

p1.x = 50;
p1.y = 125;
System.out.println("p1.x: " + p1.x);

```

É preciso usar:

```

p1.setX(50);
p1.setY(125);
System.out.println("p1.x: " + p1.getX());

```

#### 4.2.3 A referência this

Quando é preciso acessar um membro de um objeto em outra classe, usa-se uma variável que foi declarada como referência. Por exemplo, para acessar o *x* de um determinado objeto *Ponto*, usamos a referência *p1*:

```
Ponto p1 = new Ponto();
p1.x = 123;
```

Se o acesso for dentro da própria classe a variável é acessada *diretamente sem prefixo*:

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

    public double distancia() {
        return Math.sqrt(x * x + y * y);
    }
}
```

Mas existe uma referência que aponta para a instância atual. Ela é representada pela palavra reservada *this*, e pode ser usada dentro de métodos e construtores. O código abaixo é equivalente ao anterior:

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

    public double distancia() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
}
```

Neste caso particular *this* não é necessário, mas é uma boa prática usar, para deixar claro que essas são variáveis de instância e distingui-las de variáveis locais, declaradas no método. É comum, em determinadas situações (construtores e métodos *setter*) declarar variáveis locais com o mesmo nome que variáveis de instância. Neste caso, para não haver conflito é necessário prefixar a variável de instância com *this*:

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

    public void setX(int x) { // x declarada localmente no parâmetro
        this.x = x;           // x sem prefixo refere-se à variável local
    }
    ...
}
```

#### 4.2.4 Construtores

Um *construtor* é usado para inicializar uma instância, depois que a memória foi alocada para ela. O construtor é como um método, mas *tem o mesmo nome que a classe*, não tem declaração de tipo de retorno e é usado como argumento de uma instrução *new*.

As classes que criamos até agora não tiveram construtores declarados, mas eles existem. Java cria automaticamente um *construtor default sem argumentos* para toda classe que não tem construtor declarado. Portanto a classe *Ponto* tem um construtor *Ponto()*, e a classe *Linha* tem um construtor *Linha()*, que usamos na hora de construir objetos com essas classes.

As classes *PrimeiroPrograma* e *MapaDePontos* também têm um construtor default, mas eles não foram usados.

Se quisermos que algo aconteça após a construção de um objeto, podemos inserir instruções dentro do construtor. Neste caso, precisamos *declará-lo explicitamente*, da forma:

```
public class Ponto {
    public Ponto() {
        System.out.println("Um ponto acaba de ser criado!");
    }
    ...
}
```

Agora, cada vez que houver um `new Ponto()`, essa linha de texto será impressa no terminal.

É possível também declarar *um construtor com argumentos*. Por exemplo, seria mais fácil criar um ponto se pudéssemos passar o valor dos pontos diretamente, na hora de cria-los:

```
Ponto p2 = new Ponto(123, 50);
Ponto p2 = new Ponto(50, 50);
```

Para isto ser possível, temos que declarar um construtor na classe `Ponto`, que receba dois inteiros, e que copie os valores desses inteiros para os atributos `x` e `y` do ponto:

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

    public Ponto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    ...
}
```

Esta é uma situação típica onde `this` é usado para distinguir variáveis locais das variáveis de instância.

Agora é possível criar pontos da forma mostrada anteriormente, mas não é mais possível criar um ponto na posição (0,0) usando simplesmente:

```
Ponto p = new Ponto();
```

Java só fornece um construtor *default* para classes que já não declaram um. Para garantir essas duas opções, precisamos declarar (novamente) um construtor default:

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

    public Ponto(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Ponto() {}

    ...
}
```

Uma classe pode ter qualquer quantidade de construtores, desde que seja possível distingui-los pela quantidade de parâmetros e pelo tipo dos parâmetros.

### 4.3 Membros estáticos e membros de instância

Normalmente uma classe é usada para declarar atributos e métodos que serão usados por suas *instâncias*. Esses atributos e métodos só passam a existir quando o sistema alocar memória, criar os objetos e retornar as referências para as variáveis que usamos para acessar métodos e variáveis. *Eles não existem enquanto um objeto não for criado.*

Mas é possível declarar métodos e variáveis que serão sempre *associadas à classe*, que serão compartilhados por todos os objetos, e que *não precisam de uma instância*. Esses métodos e variáveis estão declarados como *static*.

Métodos e variáveis *static* podem ser chamados diretamente pelo nome, se dentro da própria classe, e usando o *nome da classe como prefixo*, se acessíveis em outra classe. Membros estáticos nunca podem ser chamados com `this`, já que `this` é uma referência que só existe em objetos instanciados, mas *podem* ser chamados com uma referência de objeto externa (embora esta seja

uma prática desaconselhada – o correto é chamar pelo nome da classe). Por exemplo, De dentro do `main()` de `PrimeiroPrograma` é possível chamar o `main()` de `MapaDePontos`:

```
public class PrimeiroPrograma {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
        MapaDePontos.main(args);
    }
}
```

### 4.3.1 Constantes

Constantes podem ser declaradas em Java usando o modificador `final`. Constantes verdadeiras são tipos primitivos ou *Strings* imutáveis, e são também `static`. É necessário que recebam um valor no momento da declaração.

Exemplos:

```
public static final double PI = 3.14159;
public static final String CODIGO_ATIVACAO = "A9e2$918m3b18Z";
```

Também pode-se declarar variáveis de instância e variáveis locais como `final`, mas elas não são realmente constantes (podem variar entre chamadas). Referências podem ser constantes (se referirem sempre a um único objeto), mas o objeto em si pode ser alterado.

### 4.3.2 Mais sobre `System.out.println()`

A classe `System` faz parte do pacote `java.lang`, assim como a classe `String` e a classe `Math`. Todas as classes desse pacote são automaticamente importados em qualquer classe Java.

`System` contém três variáveis de instância públicas, que representam streams de entrada e saída. Um deles é `out`, uma variável *estática* do tipo `java.io.PrintStream`, que redireciona tudo o que recebe para a saída padrão (o console do terminal).

A classe `PrintStream` contém vários métodos, e dentre eles vários métodos de *instância* chamados `println()`, sobrecarregados, que recebem diferentes tipos de parâmetros (`int`, `double`, `String`), etc. Nos nossos programas, usamos apenas o método que recebe `String` como argumento.

Devido ao encapsulamento, não precisamos importar nem instanciar um objeto `PrintStream` para usar seu método de instância `println()`. Isto é feito dentro da classe `System`, que disponibiliza a referência `out` publicamente. Observe também que `out` é uma variável estática (por isso não precisamos instanciar um objeto `System`).

## 4.4 Outros componentes de uma classe Java

Mesmo sem conhecer o funcionamento de uma classe, e mesmo antes de conhecer bem a linguagem Java, é possível e recomendável analisar o código de classes existentes e identificar seus componentes estruturais, sem entrar nos detalhes do que ocorre no interior dos métodos e construtores. Nesta seção apresentaremos um resumo dos principais componentes de uma classe Java que ainda não mencionamos. Eles serão explorados em detalhes mais adiante. Sabendo reconhecer esses componentes, você será capaz de abrir qualquer classe Java, identificar suas partes, e possivelmente até descobrir para que serve e como ela pode ser usada.

### 4.4.1 Convenções, espaços e endentação

Código Java bem-escrito segue convenções de nomenclatura e utiliza *espaços em branco* para organizar o código de forma a torná-lo mais legível, endentando o código contido em blocos. Se uma classe Java que você estiver analisando não estiver bem formatada, você pode sempre usar

o Eclipse e a opção Ctrl-Shift-F para reformatá-lo. Espaços e endentação a mais ou a menos não alteram o funcionamento do programa, nem afetam sua performance, e seu uso é recomendado.

O aspecto mais importante da formação é a *indentação*, pois destaca um aspecto importante da linguagem que é a visibilidade e escopo de variáveis. Blocos delimitam o escopo de variáveis (uma variável declarada dentro de um bloco só é acessível dentro do mesmo bloco ou em blocos aninhados). A endentação destaca o início e fim dos blocos, destacando o aninhamento.

Embora não haja uma regra rígida, o padrão seguido na documentação e o default em editores como o Eclipse é que blocos iniciam com o { na mesma linha, e a linha seguinte é endentada em 4 espaços, e assim por diante, voltando quatro espaços para fechar o bloco:

```
instrução {
    instrução {
        instruções
        instruções
    }
}
```

Outro aspecto importante é a escolha dos *identificadores* para nomes de variáveis, classes e métodos. A convenção é:

- *Constantes*: tudo em maiúsculas, sublinhado para separar palavras. Ex: *UMA\_CONSTANTE*
- *Pacotes*: tudo em minúsculas: *cursodejava.introducao*
- *Classes, interfaces, enums*: caixa mista, cada palavra começa com maiúscula. Ex: *NomeDaClasse*
- *Métodos e variáveis*: igual a classes, mas começando sempre com minúscula: *nomeDaVariavel, nomeDoMetodo*.
- *Parâmetro de classe*: uma letra maiúscula ou letra e número. Ex: *E, K, T, T1, T2*

É muito mais fácil entender código que segue essas convenções mínimas.

#### 4.4.2 Comentários

Comentários são trechos de texto ou código que são *ignorados pelo compilador*. Podem ser usados para *documentar* o código escrito, e também para *remover temporariamente* código que não deve ser considerado.

Existem três tipos de comentários em Java:

- Comentários de *bloco*, delimitados por /\* e \*/
- Comentários de *linha*, iniciados por // valem até o final da linha
- Comentários de bloco de *documentação*, delimitados por /\*\* e \*/ e usados antes de métodos, atributos, construtores e declarações de classe.

Comentários de linha são usados para comentar uma linha de código, ou para eliminar uma linha de código que não deve ser considerada:

```
System.out.println(); // imprime uma linha em branco
// System.out.println("não imprima");
```

Comentários de bloco são usados para comentar trechos de várias linhas. Iniciam com /\* e terminam com o primeiro \*/ encontrado. Eles não podem ser aninhados.

```
/* Este trecho inteiro foi eliminado

System.out.println("p2.y: " + p2.y);
System.out.println("p3.x: " + p3.x);
System.out.println("p3.y: " + p3.y);

*/
```

Comentários de documentação, para o compilador, são meros comentários de bloco, mas eles são usados pela ferramenta *JavaDoc* do Java para gerar documentação em HTML. Portanto é recomendado que sejam principalmente antes das classes e membros públicos de uma classe:

```
/** Esta classe é usada para construir elipses */
public class Ellipse {

    /** Calcula a área do elipse.
     * @return A área do elipse.
     */
    public double área() { ... }
}
```

#### 4.4.3 Extensão

Em Java, uma classe pode herdar a implementação de outra. Se uma classe *Fatura* herda da classe *Documento*, *Fatura* é considerada uma *subclasse* de *Documento*, e *Documento* é considerada a sua *superclasse*. Uma classe pode ter apenas uma superclasse, mas pode ter qualquer número de subclasses. A relação de herança em Java é expressa em cada subclass através da cláusula *extends*:

```
public class Fatura extends Documento { ... }
```

#### 4.4.4 Interfaces

Interfaces são classes que não têm implementação. Elas servem apenas para oferecer constantes e declarar métodos, que são implementados em outra classe. Classes que usam interfaces têm uma cláusula *implements*, que pode conter uma lista de nomes de interfaces que são implementadas naquela classe:

```
public class Pixel extends Ponto implements Desenhavel, Editavel { ... }
```

Interfaces são declaradas em arquivos *.java* usando a palavra reservada *interface*. Seus métodos não tem corpo (só são declarados):

```
public interface Desenhavel {
    void desenhar();
}
```

Desde Java 8 interfaces também podem conter métodos estáticos e métodos com implementações default.

#### 4.4.5 Enumerações

Enumerações são geralmente declaradas em arquivos *.java* individuais. A sua compilação também gera um arquivo *.class*. São usadas como constantes. Uma enumeração é declarada como *enum*:

```
public enum Estacao {
    VERAO, PRIMAVERA, INVERNO, OUTONO;
}
```

As constantes dos enums são usados em outras classes como atributos estáticos:

```
Estacao e1 = Estacao.VERAO;
Estacao e2 = Estacao.OUTONO;
```

#### 4.4.6 Anotações

Anotações são instruções e declarações especiais para determinados processadores. Elas funcionam como *meta-informação* e podem ser usados pelos processadores para gerar código e configurar ambientes. Uma anotação só tem efeito se o processador em questão reconhece-la, caso contrário elas funcionam como comentários, e são *ignoradas*.

Anotações são nomes que começam com @ e podem ou não ter parâmetros entre parênteses. Podem aparecer *antes* de classes, métodos e atributos, construtores, parâmetros de métodos e construtores. Por exemplo:

```
@Named("controle")
@SessionScoped
public class ControleBean {
    @Transactional
    public void login() { ...}
}
```

O compilador Java ignora as anotações acima, mas elas são reconhecidas por ferramentas que constroem componentes em Java EE, gerando código necessário para implantação da aplicação automaticamente.

O compilador Java também é um processador e reconhece algumas anotações como `@SuppressWarnings` (que elimina mensagens de erro moderadas), `@Deprecated` (para avisar sobre recursos obsoletos), e `@Override` (que verifica que um método está sendo redefinido). Anotações são criadas como *interfaces*.

#### 4.4.7 Classes parametrizadas

Algumas classes em Java são *containers* para outros tipos genéricos, e recebem esse outro tipo como *parâmetro*. Coleções são um exemplo típico. Nesses casos pode-se usar *classes parametrizadas*, ou *genéricas*. Uma lista de livros pode ser representado como:

```
List<Livro> lista;
```

A mesma estrutura de lista pode ser usada para organizar uma lista de *Strings*:

```
List<String> textos;
```

Algumas estruturas desse tipo recebem mais de um parâmetro. Por exemplo:

```
Map<String, Livro> fichaCatalografica;
```

#### 4.4.8 Instruções

Instruções de código Java acontecem apenas dentro de métodos, construtores e blocos *static* (usado em inicialização de variáveis estáticas). O bloco de uma classe não é uma estrutura sequencial e não pode conter instruções, exceto inicialização de variáveis.

O interior de um método pode conter instruções que (geralmente) serão executadas sequencialmente, e essas instruções podem ocorrer dentro de diferentes blocos que controlam essa execução, como blocos condicionais (`if`, `else`, `switch`), repetições (`for`, `while`), exceções (`try`, `catch`, `finally`), que podem ser aninhados. Instruções também podem incluir classes internas (declaradas e usadas dentro de métodos) e expressões lambda (a partir de Java 8).

## 5 A Biblioteca Fundamental Java (pacote `java.lang`)

A biblioteca mais importante de toda a API é `java.lang`. Todas as classes de `java.lang` são automaticamente importadas sem a necessidade de uma declaração import. A seguir estão relacionadas algumas das mais importantes classes deste pacote (consulte a documentação para maiores detalhes e outras classes):

### 5.1 Object

É a raiz de toda a hierarquia de classes em Java. Toda classe, existente ou não em Java, herda os parâmetros definidos em `Object`. É raro usar `Object` diretamente para criar instâncias. É mais

comum usar *Object* para criar subclasses, quando não se estende outra classe, ou para usá-la como referência universal, como argumentos de métodos, construtores ou vetores.

## 5.2 Class<T>

Classe que representa classes. Contém métodos para carregar classes dinamicamente e obter informações sobre elas. É muito usada em programação genérica. O objeto *Class<T>* de um objeto é obtido através do método *getClass()* de *Object*. Pode-se também representar o objeto *Class<T>* usando o nome da classe com sufixo *.class*, por exemplo:

```
Class<Ponto> clazz1 = Ponto.class;
Class<Linha> clazz2 = linha.getClass(); // linha é referência para instancia Linha
```

## 5.3 Number, Character e Boolean

*Number* é uma classe abstrata que é superclasse de *Integer*, *Long*, *Byte*, *Short*, *Float* e *Double*, que juntamente com *Character*, *Void* e *Boolean* completam a coleção de classes empacotadoras de tipos primitivos. Como os tipos primitivos em Java não são objetos, eles se beneficiam destas classes quando precisam ser usados como objetos. A conversão de objetos em primitivos e vice-versa é chamada de *autoboxing* (empacotamento) e *unboxing* (desempacotamento). É realizado de forma automática quando necessário.

As classes empacotadoras também possuem vários métodos estáticos úteis para conversão de *String* em números e booleanos. São métodos como *Double.parseDouble()*, *Integer.parseInt()*, etc. Eles recebem um *String* e tentam converter no tipo correspondente:

```
double d = new Double("12.783e-16").doubleValue();
float f = Float.valueOf("3.14159");
long k = Long.parseLong("8516962");
boolean b = new Boolean("true").booleanValue();
```

## 5.4 Math

É uma classe final que define constantes para os valores matemáticos  $\pi$  e  $e$ , além de definir um grande conjunto de funções matemáticas (métodos estáticos) para a trigonometria, exponenciação e outras operações de ponto flutuante. Também contém métodos para calcular máximos e mínimos e gerar números pseudo-aleatórios. Exemplos:

```
double d = Math.sqrt(2);
int maior = Math(7, 9);
dados = (int)(Math.random() * 6)
```

## 5.5 System e Runtime

*System* define três variáveis estáticas que representam a entrada padrão (*in*), a saída padrão (*out*) e a saída padrão de erro (*err*). Além disso, define métodos que oferecem uma interface independente de plataforma para funções do sistema. Com *System* é possível, por exemplo, executar uma aplicação externa e controlar o processo resultante, recuperar as fontes do sistema, etc. *Runtime* encapsula várias funções do sistema que são dependentes de plataforma, como a coleta de lixo, e contém vários métodos que são chamados por *System*.

## 5.6 Process, Thread e Runnable

*Process* define uma interface independente de plataforma para processos que rodam externamente ao interpretador Java. *Thread* implementa suporte para múltiplas linhas de controle rodando no mesmo interpretador Java. Para criar uma linha de controle concorrente é preciso ou estender a classe *Thread* ou implementar a interface *Runnable* e passar o objeto resultante a um construtor do *Thread*. *Thread* define métodos para controle de prioridades, interrupção e agendamento de threads. *Runnable* declara um único método: *public void run()*, que deve ser implementado pelas classes que desejarem criar linhas de execução concorrentes.

## 5.7 Throwable, Error, Exception e suas subclasses

É a classe raiz da hierarquia de erros e exceções. Objetos *Throwable* são usados nas declarações *throw*, *catch* e *finally*. É pouco comum usar a classe *Throwable*. A classe *Error* é usada para definir erros dos quais não se espera recuperar, como por exemplo, a falta de memória. *Exception*, por sua vez, já serve para definir condições excepcionais que se espera ser possível corrigir sem abandonar a execução do programa, como, por exemplo, o fato de não conseguir encontrar um determinado arquivo.

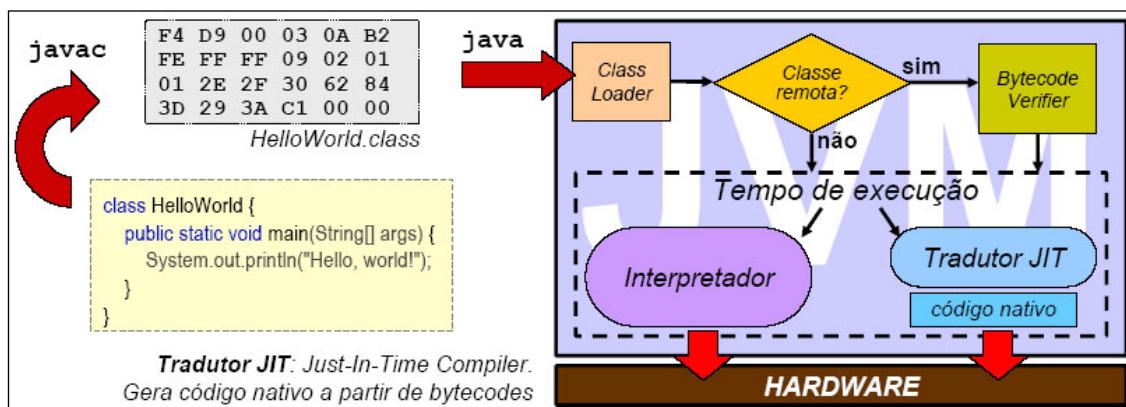
## 5.8 Cloneable

É uma interface que não contém método algum. Serve para sinalizar que o objeto criado pela classe que implementa esta interface pode ser copiado usando o método *Object.clone()*.

# 6 A máquina virtual Java (JVM)

A máquina virtual Java (*JVM - Java Virtual Machine*) é uma máquina imaginária implementada como uma aplicação de software. Ela simula um computador, com área de memória, pilha, registradores, etc. É importante conhecer um pouco do seu funcionamento para facilitar o entendimento de muitos aspectos importantes da linguagem mais adiante.

A JVM executa um tipo de código de máquina portável chamado de Java *bytecode*. Esse código é armazenado em um tipo de arquivo chamado de *class file format*. Um arquivo desse tipo é geralmente criado como resultado da compilação de código Java. A figura abaixo ilustra o processo entre a escrita de um programa em Java até a sua execução pela JVM:



Uma das decisões de *design* da plataforma Java foi a de esconder do programador detalhes da memória. Do ponto de vista de um programador Java, as áreas de memória virtual conhecidas

como a pilha e o *heap* são lugares imaginários na memória de um computador. Não interessa ao programador nem adianta ele saber onde estão nem os detalhes de como são organizados, uma vez que Java não oferece opções de escolha para alocação de memória no *heap* ou na pilha como ocorre em outras linguagens como C ou C++. Além disso, a especificação da máquina virtual garante liberdade ao implementador de máquinas virtuais Java para organizar a memória como bem entender.

## 6.1 Pilha

Um programa de computador armazena dados e instruções na memória. Instruções e valores de tamanho fixo especiais geralmente são armazenados numa estrutura de *pilha*. Por exemplo, uma seqüência de instruções e seus dados contida em um método. Quando a seqüência (método) termina de executar, geralmente esvazia-se a *pilha*. Isso tudo ocorre em baixo nível, sem que o programador Java precise se preocupar com isso.

Quando um tipo primitivo (como inteiro, caractere, ponto flutuante, booleano) de tamanho fixo é criado explicitamente através de um literal, a JVM sempre o armazena na pilha.

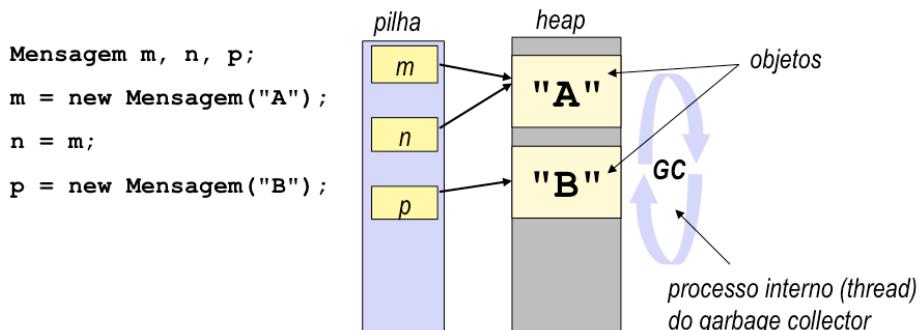
## 6.2 Heap e coleta de lixo

Outra estrutura de memória é o *heap*, que é endereçado e dinâmico. O heap permite armazenar dados que podem ser recuperados posteriormente, mesmo depois que a pilha de instruções for esvaziada. Para isso, é preciso alocar previamente o espaço necessário, e guardar o endereço do seu início. Quando a memória no heap não for mais necessária, ela deve ser liberada. Não fazer isto causa um vazamento (memory leak).

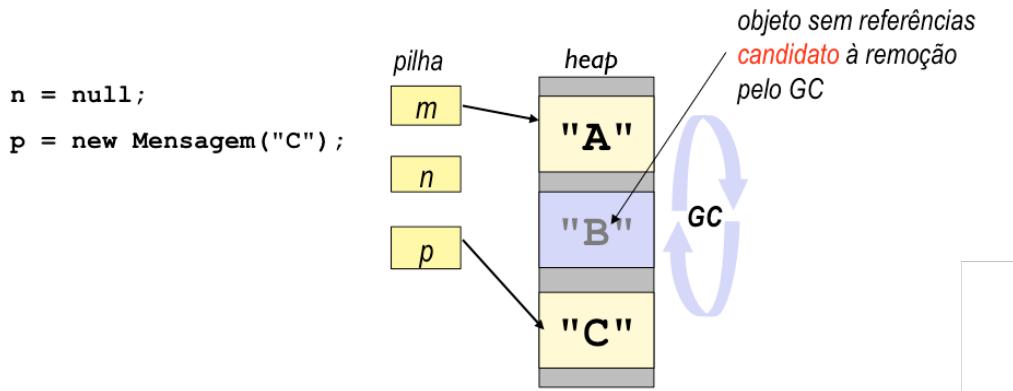
Em linguagens como C ou C++ o programador precisa explicitamente alocar e liberar memória no heap. Em Java, o programador não faz uma coisa nem outra. Quando cria um objeto, a máquina virtual Java automaticamente aloca memória no heap de acordo com o tamanho das estruturas do objeto.

A escolha da área de memória não é responsabilidade do programador. O programador também não tem o endereço dos dados no heap. Ele apenas tem uma variável que guarda essa referência que ele não pode mudar. Quando o objeto não for mais necessário, o programador pode reiniciar ou atribuir outros valores às variáveis que guardavam a referência, e o sistema se encarregará de liberar a memória alocada. Esse processo é chamado de coleta de lixo. A máquina virtual possui vários algoritmos diferentes de coleta de lixo que são usados em situações diferentes, e ela ainda pode ser otimizada se necessário.

O exemplo abaixo ilustra de forma simplificada como ocorre a alocação e liberação de memória na máquina virtual Java. Foram declaradas três variáveis de um mesmo tipo, mas apenas dois objetos foram alocados no heap. O objeto "A" possui duas referências de acesso (m,n), e o objeto "B" apenas uma (p):



Em outro momento, a referência n recebe o valor null, mas ainda é possível acessar o objeto "A" através da referência m. Mas um novo objeto é criado e sua referência é atribuída à variável p. Como resultado o objeto "B" tornou-se inacessível, e será marcado pelo algoritmo de coleta de lixo para remoção durante a próxima coleta.



## 6.3 Classloader e Classpath

Antes de um programa ser usado, a JVM carrega suas classes através de um componente chamado Classloader. O Classloader procura classes em lugares pré-definidos pelo ambiente de execução. Esses lugares são pastas ou arquivos compactados que fazem parte de uma lista chamada de Classpath.

## 6.4 Verificador de bytecode

Por último, o verificador de bytecode é usado quando classes são transferidas pela rede. Ele verifica se o bytecode não foi corrompido de alguma maneira, e se foi, impede a sua execução.

# 2 Sintaxe

---

<b>1 Identificadores</b>	<b>2</b>
<b>1.1 Palavras-chave</b>	<b>2</b>
<b>2 Literais</b>	<b>3</b>
<b>2.1 Numéricos</b>	<b>3</b>
<b>2.2 Booleanos</b>	<b>3</b>
<b>2.3 Caracteres e strings</b>	<b>4</b>
<b>3 Tipos e valores</b>	<b>5</b>
<b>3.1 Tipos primitivos</b>	<b>5</b>
<b>3.2 Arrays e objetos</b>	<b>5</b>
<b>3.3 Valores primitivos</b>	<b>7</b>
3.3.1 Valores booleanos	7
3.3.2 Caracteres	7
3.3.3 Valores inteiros	7
3.3.4 Valores de ponto-flutuante	7
<b>4 Operadores e expressões</b>	<b>8</b>
<b>4.1 Expressões de cópia e atribuição</b>	<b>9</b>
<b>4.2 Expressões numéricas</b>	<b>10</b>
4.2.1 Binárias	10
4.2.2 Unárias	10
<b>4.3 Expressões booleanas</b>	<b>10</b>
<b>4.4 Expressões de deslocamento de bits</b>	<b>11</b>
<b>4.5 Expressões com strings</b>	<b>11</b>
<b>4.6 Referências e objetos</b>	<b>11</b>
4.6.1 Igualdade	11
4.6.2 Atribuição	12
<b>4.7 Conversão e coerção (casting) de tipos</b>	<b>12</b>
<b>5 Expressões de controle de fluxo</b>	<b>13</b>
<b>5.1 Expressões condicionais</b>	<b>14</b>
5.1.1 Declaração if ... else	14
5.1.2 Declaração switch	14
5.1.3 Operador ternário	15
<b>5.2 Expressões de repetição</b>	<b>15</b>
5.2.1 Declaração while e do...while	15
5.2.2 Declaração for	15
5.2.3 Declaração for-each	16

<b>5.3 Desvios</b>	<b>16</b>
5.3.1 Return	16
5.3.2 Break e continue	16
5.3.3 Exceções	17

## 1 Identificadores

Identificadores em Java são os nomes que usamos para identificar variáveis, classes, e métodos. Não fazem parte da linguagem e são criados arbitrariamente pelo programador.

Pode-se usar qualquer letra ou dígito do alfabeto *Unicode*, ou os símbolos “\$” e “\_” (sublinhado) em um identificador. É illegal iniciar identificadores com um número. Dígitos podem ser usados como parte de identificadores desde que não sejam o primeiro caractere.

Estes são alguns identificadores legais em Java:

```
ping_pong    item    Filme    i    F2    PARTE_1    $indice    бутерброд    зал
```

(Os últimos dois identificadores acima *não* começam com número, mas com os caracteres “б” (b) e “з” (z) do alfabeto cirílico, portanto são legais).

Esses outros identificadores são ilegais:

```
ping-pong    Johnson&Johnson    R$13.00    2aParte
```

Identificadores em Java podem ter qualquer tamanho. Não há limitação em relação ao número de caracteres que podem ter, porém deve-se evitar nomes muito grandes para identificar classes, uma vez que também são usados como nomes de arquivo. Palavras reservadas (veja adiante) não podem ser usadas como identificadores.

Java distingue letras maiúsculas de minúsculas. Isto também vale para palavras usadas como identificadores. *Valor* é diferente de *valor* que é diferente de *VALOR*.

O ideal é usar nomes claros, explicativos, mas evitar caracteres acentuados e outros alfabetos. Limitar identificadores aos caracteres ASCII é uma boa prática. Embora as seguintes palavras *sejam* identificadores legais em Java:

```
Сәо    variável    CLASS    R$13    índice    фильм    קוד ייחודי    中文字
```

Não são recomendadas, pois dificultam a leitura, podem confundir e requerem a existência de fontes para exibição que podem não estar presentes em todos os sistemas. Já pensou se você usa a palavra *índice* como variável e depois, por engano, usar *indice*? É um erro bem difícil de achar. *CLASS* não é palavra reservada, mas *class* é. Usar *Фильм* pode ser interessante se você fala russo, mas imagine que este seja o nome de uma classe executável. Em um computador sem as fontes para exibir cirílico aparece como ??????. Como você vai conseguir rodar o programa se não tiver as fontes correspondentes no seu sistema?

### 1.1 Palavras-chave

As palavras seguintes, junto com os valores *true* e *false*, que são *literais booleanas*, são reservadas em Java. Não podem ser usadas para identificar classes, variáveis, pacotes, métodos ou serem usadas como nome de qualquer outro identificador.

abstract	continue	for	new	switch
----------	----------	-----	-----	--------

<code>assert</code>	<code>default</code>	<code>goto</code>	<code>package</code>	<code>synchronized</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>private</code>	<code>this</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>protected</code>	<code>throw</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>throws</code>
<code>case</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>transient</code>
<code>catch</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>try</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>strictfp</code>	<code>volatile</code>
<code>const</code>	<code>float</code>	<code>native</code>	<code>super</code>	<code>while</code>

Apesar de reservadas, as palavras `goto` e `const` não são usadas.

Os nomes de classes fundamentais da linguagem (classes do pacote `java.lang`) não são palavras reservadas, mas você deve evitar usá-las já que causam erros de ambiguidade.

## 2 Literais

Literais são valores. Podem ser valores numéricos, booleanos, caracteres individuais ou cadeias de caracteres. Literais podem ser usados diretamente em expressões, passados como argumentos em métodos ou atribuídos a uma variável. Exemplos de literais são:

- `12.4` (12,4 ponto flutuante decimal)
- `0377` (377 inteiro octal)
- `0xff991a` (FF991A inteiro hexadecimal)
- `true` (literal booleana)
- `'a'` (caractere)
- `"barata"` (cadeia de caracteres)

### 2.1 Numéricos

Os literais numéricos são representados em formato decimal, por default. Se um literal numérico for precedido por um zero, será considerado um número octal, e se for precedido por 0x, será interpretado como um número hexadecimal. Veja alguns exemplos de literais numéricos:

```
12.4  0xab779c .27777e-23  0137   20L  Double.NaN  123_456_789  F_000  0B101
```

Literais numéricos variam em faixa e representação dependendo do *tipo* de dados das variáveis usadas para armazená-los. Veja na seção seguinte as faixas de valores suportados para cada tipo.

### 2.2 Booleanos

Os literais booleanos são apenas dois: `true` e `false`. Representam uma condição verdadeira ou falsa.

## 2.3 Caracteres e strings

Os literais de caracteres podem ser representados através de um caractere isolado entre aspas simples (por exemplo: 'a', 'Z') ou usando uma seqüência de escape especial (veja adiante), também entre aspas simples. Veja alguns exemplos:

```
'H'   '\n'   '\u0044'   '\u3F07'   '1'   '\\'
```

Apesar do tipo String não ser um tipo primitivo em Java (String é uma classe que representa objetos do tipo “cadeia de caracteres”), Java define literais do tipo String formados por conjuntos de caracteres entre aspas duplas. Alguns exemplos:

```
"anta"      "vampiros são morcegos"      ""      (vazia)
"\u3F07\u3EFA \u3F1C"      " "      (espaço)
"Uma linha\nDuas Linhas\tTabulação"
"Жава – очень хороший язык программирования!"
```

Qualquer caractere em Java pode ser representado usando o padrão Unicode, de 16 bits, que permite representar 65536 caracteres diferentes. Além disso, as sequências de escape listadas na tabela a seguir podem ser usadas para representar certos valores especiais que podem aparecer em uma literal do tipo char ou String.

Seqüência	Valor do Caractere
\b	Retrocesso (backspace)
\t	Tabulação
\n	Nova linha (new line)
\r	Retorno de carro (carriage return)
\"	Aspas
\'	Aspa
\\\	Contra Barra
\nnn	O caractere correspondente ao valor octal nnn, onde nnn é um valor entre 000 e 0377.
\unnnn	O caractere Unicode nnnn, onde nnnn é de um a quatro dígitos hexadecimais. Seqüências Unicode são processadas antes das demais seqüências.

Os caracteres de escape Unicode são formas de representar caracteres que não podem ser exibidos em sistemas que não suportam Unicode ou que não possuem as fontes corretas para exibir os caracteres. Por exemplo, em um terminal Unicode, você pode ver na tela os ideogramas Higarana e Katakana (japonês) e vários outros que correspondem à faixa \u3040 a \u9FFF do código Unicode. Em um sistema ASCII, Java representa esse caractere usando o escape Unicode \unnnn.

Para representar aspas e contra-barras dentro de um literal String ou de caractere, é necessário usar contra-barras como escapes (não é possível, como em outras linguagens, usar apóstrofes). Por exemplo, para imprimir a linha:

```
"Doom II", C:\games\doom
```

é preciso usar:

```
System.out.println("\"Doom II\"", C:\\games\\doom");
```

## 3 Tipos e valores

Uma variável em Java *sempre* possui um tipo bem definido, que é determinado no momento que a variável é declarada. Antes de ser usada a variável precisa ser inicializada com um valor, que pode ser igual ou *compatível* com o tipo declarado. Se for uma variável local (declarada dentro de um método ou construtor), a inicialização precisa ser explícita. Variáveis de instância são inicializadas automaticamente com valores *default*. Se o tipo for uma classe ou array, o valor default é null. Se for um tipo primitivo, o default é 0 se for um número, o caractere \u0000 se for char, e o valor false se for boolean.

### 3.1 Tipos primitivos

Tipos primitivos representam valores escalares e podem ser booleanos, caracteres, inteiros ou ponto-flutuante. Tipos primitivos são *armazenados na pilha da JVM* e tem tamanho fixo. Os tipos primitivos do Java, seu tamanho em bytes e a faixa de valores que representam estão listados na tabela abaixo.

Tipo de dados	Tamanho	Faixa de valores
boolean	8 bits	true ou false
byte	8 bits	-128 a 127
short	16 bits	-32768 a 32767
char	16 bits	\u0000 a \uffff
int	32 bits	-2147483648 a 2147483648
long	64 bits	-9223372036854775808 a 9223372036854775807
float	32 bits	1,40239846e-45 a 3,40282347e38
double	64 bits	4,94065645841264544e-324 a 1.79769313486231570e308

Uma variável declarada como um tipo primitivo pode conter valores compatíveis com aquele tipo:

```
byte b1 = 100;      // É compatível, pois 100 cabe em um byte
int numero = b1;    // Copiando valor de b1 (100) para numero - é compatível,
                   // pois um byte cabe em um int
b1 = numero;        // Apesar de 100 caber em byte, o conteúdo de um int pode
                   // não caber em um byte esta linha dá erro de compilação!
```

### 3.2 Arrays e objetos

Arrays e objetos diferem dos tipos primitivos porque têm tamanho variável e são armazenados no heap – uma área de memória alocada aleatoriamente. As variáveis declaradas como arrays e

objetos armazenam na pilha o *endereço* do array ou objeto. Esse endereço é inacessível e imutável, e é usado apenas como *referência* para ligar a variável ao objeto.

Arrays são coleções de valores. Para declarar uma variável como um array, usa-se o tipo dos elementos do array seguido de []:

```
int[] valores; // um array de ints
double[] precos; // um array de doubles
Point[] pontos; // um array de objetos do tipo Ponto
```

Essa declaração não inicializa o array, que precisa informar o número de elementos. A inicialização é feita com o operador new, que aloca memória no heap e retorna uma referência:

```
valores = new int[5]; // um array contendo 5 inteiros (todos com valor zero)
precos = new double[3]; // um array com 3 doubles (todos com valor 0.0)
pontos = new Point[10]; // um array com capacidade de conter 10 objetos do tipo
                        // Point (mas que só contém 10 referências com valor null)
```

Objetos são declarados usando classes:

```
Point ponto;
String texto;
Integer num;
```

E também são inicializados usando o operador new, que retorna uma referência. A única exceção é para objetos do tipo String e *classes empacotadoras de tipos primitivos*, que podem ser inicializados diretamente, através de um literal:

```
ponto = new Point();
texto = new String("Era uma vez..."); // usando operador new
texto = "Era uma vez..."; // usando literal
num = new Integer(5); // usando operador new
num = 5; // usando literal (boxing automático)
```

Todos os tipos primitivos possuem classes empacotadoras (wrappers) correspondentes que empacotam os valores primitivos para que possam ser usados em lugares onde apenas objetos podem ser usados. A conversão, chamada de *auto-boxing*, é automática. A tabela abaixo relaciona os tipos primitivos e as classes correspondentes.

<b>Tipo primitivo</b>	<b>Classe empacotadora</b>
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

Uma referência pode ser copiada, atribuindo seu valor (o endereço) para outra variável que tenha sido declarado com um tipo de dados compatível. Com isso, as duas referências apontarão para o mesmo objeto. Esse comportamento é diferente do que ocorre quando tipos primitivos

são copiados. Como eles são armazenados diretamente na pilha, a cópia causa a duplicação do valor.

### 3.3 Valores primitivos

#### 3.3.1 Valores booleanos

Valores booleanos não são números e nem podem ser tratados como tal. Também não acontece conversão automática de valores booleanos em números (como ocorre em outras linguagens), ou seja, 0 e false não são a mesma coisa. A conversão em Java, se necessário, sempre precisa ser explícita

```
boolean b; int i;
b = (i != 0); // converte 0 para false e !0 para true
i = b ? 1 : 0; // converte true para 1 e false para 0
```

#### 3.3.2 Caracteres

Valores do tipo *char* contém um caractere Unicode de dois bytes. A conversão de *char* em *int* ou *long* (ex: via atribuição) obtém o valor Unicode do caractere. Um *char* sempre pode ser manipulado como um *int*. Se for necessário representar o caractere, porém, é preciso convertê-lo (através de uma operação de cast) para *char*:

```
int letra = 'A';
System.out.println("O código Unicode da letra "
+ ((char)letra) + " é " + letra);
```

#### 3.3.3 Valores inteiros

Todos os inteiros em Java tem sinal. Inteiros podem ser do tipo *byte* (8 bit), *short* (16 bit), *int* (32 bit) ou *long* (64 bit).

A representação default para inteiros é *int*. Se uma atribuição para uma variável do tipo *long* recebe um *int*, este é automaticamente convertido (e ampliado) para *long*. Se o literal for um número maior que o valor máximo de um *int*, porém, o número será *long* e provocará erro ao ser atribuído a uma variável do tipo *int*.

Pode-se distinguir literais do tipo *long* das constantes *int* utilizando o caractere “L” como sufixo (ex: 127L). Evite, porém, usar o “L” minúsculo (é permitido, mas como ele se parece muito com o algarismo “1”, poderá ser fonte de confusão).

#### 3.3.4 Valores de ponto-flutuante

Qualquer valor numérico contendo um ponto decimal ou um expoente (caractere “e” seguido de um número) é um literal de ponto flutuante. São sempre do tipo *double* (64 bit – dupla precisão) por default. Tipos *float* (32 bits – precisão simples), quando inicializados, devem conter o sufixo “F” pois sem esse sufixo, o literal é considerado *double*. O sufixo “D” pode, opcionalmente, ser usado para identificar valores de dupla-precisão:

```
float f = 300.0;           // Erro: 300.0 é double e não cabe em float
float f = 300.0f;          // OK pois 300.0f é float
float f = (float) 300.0;   // OK. double convertido em float usando cast
float f = 300;             // OK. int converte em float sem cast
```

Divisões por zero provocam números infinitos ou indeterminações identificadas pelas constantes *Float.POSITIVE\_INFINITY*, *Float.NEGATIVE\_INFINITY* e *Float.NaN*, que representam respectivamente, infinito positivo, infinito negativo e “não é número”. Este comportamento é diferente do que ocorre com valores inteiros, onde uma divisão por zero provoca uma exceção (*ArithmaticException*).

## 4 Operadores e expressões

Operadores permitem a realização de tarefas como adição, subtração, multiplicação, atribuição, acesso, etc. Podem ser divididos em operadores booleanos, operadores de atribuição, operadores numéricos, operadores de objetos e operadores lambda.

A tabela abaixo lista os operadores usados na linguagem Java:

Operador	Função	Operador	Função
+	Adição	~	complemento
-	Subtração	<<	deslocamento à esquerda
*	Multiplicação	>>	deslocamento à direita
/	Divisão	>>>	desloc. a direita com zeros
%	Resto	=	atribuição
++	Incremento	+=	atribuição com adição
--	Decremento	-=	atribuição com subtração
>	Maior que	*=	atribuição com multiplicação
>=	Maior ou igual	/=	atribuição com divisão
<	Menor que	%=	atribuição com resto
<=	Menor ou igual	&=	atribuição com AND
==	igual	=	atribuição com OR
!=	não igual	^=	atribuição com XOR
!	NÃO lógico	<<=	atribuição com desloc. esquerdo
&&	E lógico	>>=	atribuição com desloc. direito
	OU lógico	>>>=	atrib. C/ desloc. a dir. c/ zeros
&	AND	? :	Operador ternário
^	XOR	(tipo)	Conversão de tipos (cast)
	OR	instanceof	Comparação de tipos
.	Acessa membros de um objeto/classe	, ; :	Operadores de expressões <i>for</i>
->	Seta (lambda)	::	Referência (lambda)

São vários tipos diferentes de operadores: atribuição, adição, multiplicação, comparação, deslocamento, etc. A ordem em que uma expressão é resolvida depende da precedência dos seus operadores, que é mostrada na tabela a seguir.

Os valores em posição mais alta na tabela têm maior precedência. Na posição horizontal, a precedência é a mesma. A primeira coluna indica a associatividade. D a E significa Direita para Esquerda.

Assoc	Tipo de Operador	Operador
D a E	separadores	[] . ; , () : -> ::
E a D	operadores unários	new (cast) ++expr --expr +expr -expr ~ !
E a D	multiplicativo	* / %
E a D	aditivo	+ -
E a D	deslocamento	<< >> >>>
E a D	relacional	< > >= <= instanceof
E a D	igualdade	== !=
E a D	AND	&
E a D	XOR	^
E a D	OR	
E a D	E lógico	&&
E a D	OU lógico	
D a E	condicional	? :
D a E	atribuição	= += -= *= /= %= >>= <<= >>>= &= ^= !=
E a D	Incr. pos-fixado	expr++ expr--

## 4.1 Expressões de cópia e atribuição

A atribuição é realizada com o operador `=`, que serve apenas para atribuição. A *comparação* é realizada com um sinal de `=` duplo. A atribuição copia o *valor* da variável ou constante do lado direito para a variável do lado esquerdo.

```
x = 13;      // copia a constante inteira 13 para x
y = x;        // copia o valor contido em x para y
```

A atribuição copia valores:

- O valor armazenado em uma variável de tipo primitivo é o valor do número, caractere ou literal booleana (true ou false)
- O valor armazenado em uma variável de tipo de classe (referência para objeto) é o ponteiro para o objeto ou null.

Conseqüentemente, copiar referências por atribuição não copia objetos mas apenas cria novas referências para o mesmo objeto!

A atribuição podem ser combinada com uma operação usando os operadores `+=`, `-=`, `*=`, `/=`, `%=`. A expressão `x = x + 1` equivale a `x += 1`.

## 4.2 Expressões numéricas

### 4.2.1 Binárias

Operações binárias são as operações numéricas mais comuns. Elas ocorrem com base na precedência que determina em que ordem serão realizadas. Por exemplo, operações de multiplicação são realizadas antes de operações de soma:

```
int x = 2 + 2 * 3 - 9 / 3; // 2+6-3 = 5
```

Parênteses podem ser usados para sobrepor a precedência:

```
int x = (2 + 2) * (3 - 9) / 3; // 4*(- 6)/3 = - 8
```

Mas a maior parte das expressões de mesma precedência é calculada da esquerda para a direita:

```
int y = 13 + 2 + 4 + 6; // (((13 + 2) + 4) + 6)
```

As expressões causam promoção e conversão de tipos. Qualquer tipo menor que int é automaticamente convertido para int, ou, se houver um tipo maior ou me maior precisão na expressão, todos os tipos serão convertidos para ele.

### 4.2.2 Unárias

Usar `-` antes de um número muda o seu sinal. O `+` também pode ser usado, mas ele não altera nada. Operadores unários podem ser misturados a operações binárias porque eles têm precedência. Por exemplo, a expressão abaixo é legal:

```
int x = 13 + -12;
```

Também são bastante usados os operadores unários `“++”` e `“–”` para incrementar (somar 1) ou decrementar (subtrair 1), respectivamente, uma variável. Estes operadores podem alterar o valor da variável *antes* ou *depois* que ela for usada, e sua precedência varia. Se aparecem antes, tem a maior precedência. Se aparecem depois, tem a menor (são incrementadas até mesmo depois da atribuição) Por exemplo:

```
int a = 10, b = 5;
int x = ++a;    // a contém 11, x contém 10
int y = --b;    // b contém 4, y contém 4
```

A atribuição foi feita DEPOIS!

```
int x = a++;    // a contém 11, x contém 10
int y = b--;    // b contém 4, y contém 5
```

A atribuição foi feita ANTES!

## 4.3 Expressões booleanas

Os operadores relacionais, como `<`, `>`, `=`, etc. e os operadores lógicos como `&&`, `||`, sempre produzem um resultado booleano true ou false.

Os relacionais compararam os valores de duas variáveis ou de uma variável e uma constante. Os operadores `==` e `!=` também podem ser usados como objetos para verificar se são o mesmo objeto ou se ocupam referências diferentes.

Os resultados de expressões relacionais podem ser concatenados com operadores lógicos e formar expressões mais longas. Por exemplo:

```
Boolean b = (3 > x) && !(y <= 10);
```

A expressão será realizada até que o resultado possa ser determinado de forma não ambígua. Este efeito é chamado de curto-círcuito (*short-circuit*). Ou seja, se a primeira expressão resultar em false, e ela estiver concatenada com o resto da expressão através e um *and* lógico &&, já é possível saber que o resultado final será false, e assim a expressão toda não precisará ser calculada. Isto permite que se faça testes em Java que causariam erro de compilação em outra situação:

```
if(obj != null && obj.toString().equals("abc")) { ... }
```

Por causa do efeito de curto-círcuito, apenas o *obj != null* é testado se o valor de *obj* for *null*. Isto evita o *NullPointerException* da expressão seguinte, que acessa um membro de *obj*.

Os operadores orientados a bit &, |, , usados para realizar operações AND, OR, XOR em números, bit por bit, também podem ser usados como operadores lógicos, mas eles não têm esse efeito de curto circuito. Eles normalmente são usados com inteiros e resultados são números inteiros. Suportam atribuição conjunta usando &=, |= ou ^=.

## 4.4 Expressões de deslocamento de bits

Os operadores de deslocamento de bits servem para manipulação em baixo nível de estruturas compactas como cabeçalhos de protocolos, determinados arquivos binários, etc. e alteram valores através de operações binárias de deslocamento. Eles são:

- << deslocamento de bit à esquerda multiplicação por dois)
- >> deslocamento de bit à direita divisão truncada por dois)
- >>> deslocamento à direita sem considerar sinal (acrescenta zeros)

Eles operam sobre inteiros e inteiros longos. Tipos menores (short e byte) são convertidos a int antes de realizar operação. Podem ser combinados com atribuição através dos operadores: <<=, >>= ou >>>=.

## 4.5 Expressões com strings

Java reusa o operador “+” para realizar a concatenação de Strings, além da adição de valores numéricos. Em uma operação usando “+” com dois operandos, se um deles for String, o outro será convertido para String e ambos serão concatenados.

A operação de concatenação, assim como a de adição, ocorre da direita para a esquerda

```
String s = 1 + 2 + 3 + "=" + 4 + 5 + 6;
```

O resultado da expressão acima será:

```
"6=456"
```

## 4.6 Referências e objetos

### 4.6.1 Igualdade

O operador == testa apenas a igualdade entre dois *valores*. Se os valores forem tipos primitivos como números, booleanos, etc. de tamanho fixo que são armazenados na pilha, o teste é adequado. Se forem objetos, o que está sendo testado é apenas que os dois objetos ocupam o mesmo endereço de memória, o que pode ou não ser verdade.

É possível que dois strings idênticos ocupem endereços diferentes. Se forem testados com `==` o resultado da expressão será false.

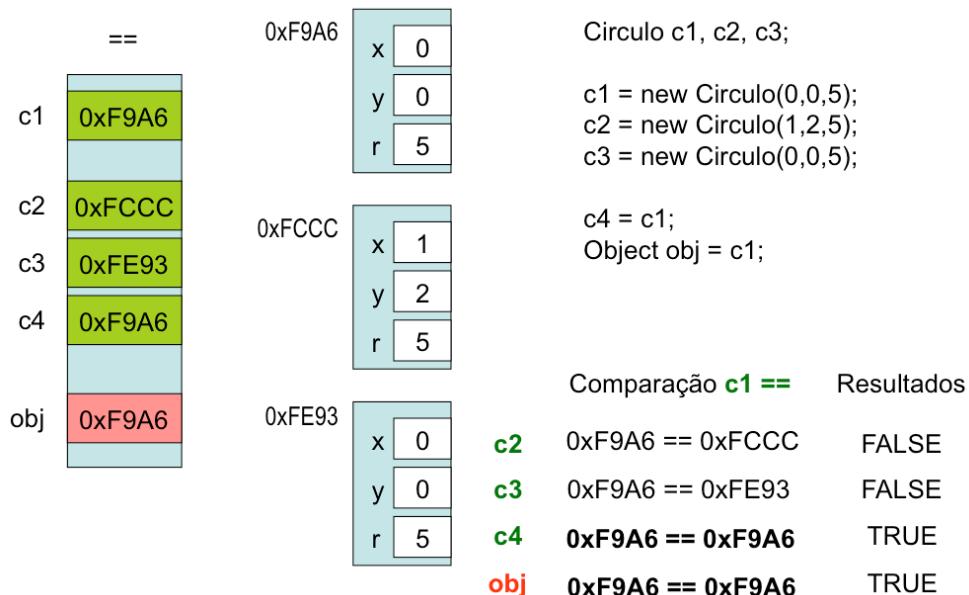
Todos os objetos possuem um método `equals()` herdado de `Object`. Objetos novos devem redefinir `equals()` com uma implementação que represente critérios de igualdade para objetos da classe. Objetos da API Java e de bibliotecas bem escritas já implementam `equals()` corretamente. A implementação de `equals()` em `String` verifica se dois strings possuem exatamente os mesmos caracteres na mesma ordem, independente de serem o mesmo objeto.

Portanto, objetos só devem ser testados com `equals()`, nunca com `==`.

#### 4.6.2 Atribuição

A atribuição também copia apenas o valor da referência. Se usado com tipos primitivos, é possível copiar uma variável e alterar a original, sem que a copiada seja afetada. Com objetos isto não vai acontecer. O valor copiado é a referência, que é imutável. E se, através dessa referência, um objeto for alterado, todos os que também tiverem referência para ele irão perceber a mudança.

O desenho abaixo ilustra uma situação com três objetos diferentes e cinco referências (sendo uma de um tipo compatível, mas diferente do objeto criado).



Para copiar um objeto é preciso ou criar um novo e copiar todos os campos primitivos para o objeto novo, ou implementar o método `clone()`, que pode ser chamado em objetos para fazer cópias automaticamente:

```
Circulo novo = c4.clone(); // Cria a cópia de um objeto em nova referência.
```

#### 4.7 Conversão e coerção (casting) de tipos

Qualquer tipo numérico pode ser convertido para outro tipo numérico durante uma atribuição. Apenas os booleanos não são conversíveis. As conversões automáticas (implícitas) só podem ocorrer quando o tipo que recebe a atribuição tem o mesmo tamanho em bits ou precisão, ou mais que o tipo atribuído. No contrário, o compilador irá reclamar.

Para fazer essas conversões “ilegais”, pode-se usar o operador de coerção (cast). Com ele, o risco fica com o programador que declara, com o cast, a possibilidade de perder dados. Um cast ou coerção é expresso informando o tipo desejado entre parênteses, durante a atribuição:

```
int numero = 100;
byte b = (byte) numero;
```

O compilador não deixa que numero seja atribuído a b, porque ints não cabem em bytes, mas o cast faz com que o compilador ignore essa regra, e fica sob a responsabilidade do programador garantir que o conteúdo de numero realmente caiba em b. Se não couber, haverá um erro de overflow e o valor resultante será incorreto.

Há três tipos de conversões entre tipos primitivos:

- Por atribuição ou passagem de parâmetro em método,
- Por promoção aritmética
- Por coerção (casting)

A conversão por *atribuição* é legal quando o tipo do lado esquerdo é maior em número de bytes ou e precisão que o tipo do lado direito:

```
int i = 10;
double d = 12.3;
d = i; // legal, d armazena 10.0
i = d; // ilegal!!!
```

A conversão por passagem de parâmetro em um método é equivalente. Um método declara receber variáveis de um determinado tipo e recebe outro. Se o tipo que o método for receber for menor que o do método, a conversão também é legal.

```
public void agua(int quantidade) { ... }
(...)
char c = 'A';
float f = 1.1f;
x.agua(c);           // OK
x.agua(f) {...}      // ilegal
```

A conversão por promoção aritmética ocorre quando há uma expressão com vários tipos diferentes. Antes da expressão ser calculada, todos os tipos são promovidos para o tipo maior, ou seja, na expressão abaixo, todos serão promovidos para double:

```
int x = 5;
long ab = 10;
double = x + ab + 3.25; // default de ponto-flutuante é double
```

Finalmente, com coerção o programador tem a disposição um meio de converter qualquer tipo. O programador é que terá que resolver se o resultado é válido.

```
int i = 10;
double d = 12.3;
d = i; // legal, d armazena 10.0
i = d; // ilegal!!!
i = (int)d; // legal, mas trunca o resultado.
```

A promoção aritmética também ocorre em operações entre bytes e shorts. Com estes valores, eles são sempre promovidos para int antes de operarem.

```
byte b = b1 + b2; // b1 e b2 são byte, mas operação + converte para int - illegal!!!
int i = b1 + b2; // legal, porque atribuição é para int
byte b = (byte) (b1 + b2); // legal por causa do cast
```

## 5 Expressões de controle de fluxo

Estas expressões controlam o fluxo de controle de um procedimento.

## 5.1 Expressões condicionais

Java possui três formas de escrever expressões condicionais:

- Declaração if-else
- Declaração switch
- Operador ternário

### 5.1.1 Declaração if ... else

*Sintaxe básica:*

```
if (expressão) {...}
    else if (expressão) { .. }
    ...
    else {...}
```

O primeiro if é obrigatório. Os outros blocos são opcionais.

A expressão precisa produzir um resultado booleano (true ou false). Se resultar em true, o código dentro do primeiro bloco entre { e } será executado, caso contrário o bloco else é executado, se existir, e assim por diante. Exemplo:

```
if (valor == 0) {
    fatorial = 1;
} else {
    fatorial = valor;
    ...
}
```

### 5.1.2 Declaração switch

*Sintaxe:*

```
switch (expressão) {
    case constante_1 :
        instruções;
        break;
    case constante_2 :
        instruções;
        break;
    ...
    case constante_n :
        instruções;
        break;
    default:
        instruções;
        break;
}
```

A declaração switch recebe uma expressão que produz um resultado numérico, String ou enumeração (enum) e compara o valor com valores constantes em cada uma das cláusulas case, em ordem. A primeira que combinar é executada. Depois disso, o controle deixa o switch quando encontrar um break (se não houver break, ele continua executando o switch, inclusive os outros cases). Se nenhum dos cases combinar com o valor recebido, a cláusula default, se existir, será executada. A cláusula default deve ser a última, para que os cases sejam testados.

Exemplo:

```
switch(estacao) {
    case "primavera": // se entrar aqui, continua duas estacoes
        System.out.println("Primavera");
    case "verao":
        System.out.println("Verão");
```

```

        break;
    case "outono": // se entrar aqui, continua duas estacoes
        System.out.println("Outono");
    case "inverno":
        System.out.println("Inverno");
        break;
    default:
        System.out.println("Viagem cancelada");
}

```

O break é essencial para sair de uma cláusula case. A única exceção é se houver um return ou ocorrer uma exceção (Exception) no código.

### 5.1.3 Operador ternário

*Sintaxe:*

```
teste ? valor se true : valor se false
```

Este operador realiza uma expressão condicional geralmente dentro de uma atribuição, com três termos, separados por ? e :.

- um *teste*, que deve retornar valor booleano, ?
- um bloco que será executado se a expressão for true, :
- um bloco que será executado se a expressão for false.

O valor produzido no segundo ou terceiro bloco será atribuído à variável que recebe o resultado.  
Exemplos:

```
double desconto = quantidade > 10 ? 20.0 : 10.0;
System.out.println(quantidade + " produto" + (quantidade == 1 ? "" : "s"));
```

## 5.2 Expressões de repetição

### 5.2.1 Declaração while e do...while

*Sintaxe básica:*

```
while (expressão) {...}
```

ou

```
do {...} while (expressão);
```

A expressão é testada a cada repetição e deve ter um resultado true ou false. Enquanto o resultado da expressão for true, o bloco de while será executado.

Na declaração do... while, o bloco é executado pelo menos uma vez antes da expressão ser testada.

*Exemplo:*

```
while (valor <= maxValor) {
    ...
    valor++;      // valor = valor + 1
} // fim do while
```

### 5.2.2 Declaração for

*Sintaxe básica:*

```
for (inicialização; teste; incremento) {...}
```

As expressões inicialização, teste e incremento controlam o loop. O teste deve ser uma expressão booleana que será testada a cada repetição. A inicialização é executada uma vez. Pode ser uma ou várias expressões separadas por vírgula. O incremento é executado no final de cada repetição

(depois que todo o bloco foi executado), e pode conter uma ou mais expressões separadas por vírgula. As três expressões são opcionais. Se forem todas omitidas, o loop será infinito.

*Exemplo:*

```
for (int parte = 10; parte > 1; parte--) {
    fatorial = fatorial * parte;
}
```

### 5.2.3 Declaração for-each

*Sintaxe básica:*

```
for (Tipo variavel : colecao) {...}
```

Este for é usado para iterar por vários elementos de uma coleção. Funciona com coleções e arrays. O tipo da variável deve ser compatível com os elementos da coleção. O loop irá repetir o bloco para cada elemento da coleção.

*Exemplo:*

```
String[] cidades = {"São Paulo", "Paris", "Rio de Janeiro", "Moscou"};
for(String cidade : cidades) {
    System.out.println(cidade);
}
```

## 5.3 Desvios

São instruções que redirecionam o controle para outro lugar. Há três maneiras de realizar desvios em Java:

- Retorno antecipado de um método, usando return antes do final
- Lançamento de uma exceção
- Break e continue em loops

### 5.3.1 Return

Uma instrução return, ocorrendo em qualquer lugar e dentro de qualquer bloco, força imediatamente o retorno do método à instrução que o chamou. Se o método dentro do qual o return ocorre declara void como tipo de retorno, a instrução return não deve ter argumentos, caso contrário ela deve retornar o tipo declarado.

*Exemplo:*

```
public void teste() {
    if(hoje.hora > 12) {
        System.out.println("Tarde demais!");
        return;
    }
    // continua ...
}
```

### 5.3.2 Break e continue

Uma instrução break pode ser usada dentro de um loop para forçar a saída, e o fim desse loop. No exemplo abaixo, se o índice da repetição for igual a um número gerado aleatoriamente, o controle sairá do loop:

```
for(int i = 0; i < 10; i++) {
    int azar = (int)(Math.random() * 10);
    if(i == azar) {
        System.out.println("Deu azar! Caindo fora!");
        break;
}
```

```

        System.out.println("Linha " + i);
    }
}

```

Para pular uma repetição (mas continuar no loop) pode-se usar a instrução `continue`. Neste outro exemplo, se o número do índice coincidir com o número gerado aleatoriamente, uma variável é incrementada e o resto da iteração é ignorada (o número da linha não será impresso). Mas o loop continua na iteração seguinte:

```

int iguais = 0;
for(int i = 0; i < 10; i++) {
    int sorte = (int)(Math.random() * 10);
    if(i == sorte) {
        ++iguais;
        continue;
    }
    System.out.println("Linha: " + i); // linha sera ignorada se continue executar
}
System.out.println("Elementos iguais: " + iguais);

```

Se houver vários loops aninhados, o `break` ou `continue` saem ou pulam uma iteração apenas do loop onde ocorrem. Se a intenção for sair de outro loop, é preciso rotular o loop com um label, e chamar o `break` ou `continue` com parâmetro:

```

externo:
for( ...) {...}
    for( ... ) { ...
        continue externo; // sai de ambos os loops
    }
}

```

### 5.3.3 Exceções

Exceções em Java ocorrem quando uma instrução `throw` é executada. Funciona como um `return`, e retorna para o método que a chamou, mas não volta a executar a linha seguinte de código, retorna novamente até o primeiro método da cadeia, e encerra o programa, a menos que algum método pelo caminho capture a exceção.

Muitas vezes essa instrução ocorre no interior de um método, que como é encapsulado não é visível ao código que chama o método, mas métodos que potencialmente causam exceções que devem ser tratadas, as declaram como parte de sua assinatura (cláusula `throws`). Uma exceção indica uma condição excepcional, um erro, e deve ser capturada e tratada.

Exceções serão tratadas em um capítulo a parte.

# 3 Classes e objetos

---

<b>1 Classes</b>	<b>2</b>
<b>1.1 Membros de instância e membros estáticos</b>	<b>3</b>
<b>1.2 Atributos (campos de dados)</b>	<b>3</b>
1.2.1 Modificadores	4
1.2.2 Valor inicial	4
1.2.3 Anotações	4
<b>1.3 Métodos</b>	<b>5</b>
1.3.1 Modificadores	5
1.3.2 Parâmetros	6
1.3.3 Varargs	6
1.3.4 Variáveis locais	6
1.3.5 Exceções	7
1.3.6 Anotações	7
<b>1.4 Construtores</b>	<b>8</b>
<b>1.5 Sobrecarga de nomes</b>	<b>8</b>
<b>1.6 Blocos de inicialização</b>	<b>9</b>
<b>1.7 Herança</b>	<b>9</b>
1.7.1 Como estender uma classe	10
1.7.2 Construtores, super() e this()	11
1.7.3 Classes finais	12
<b>1.8 Encapsulamento e modificadores de acesso</b>	<b>13</b>
1.8.1 Interface privativa ao pacote	13
1.8.2 Interface para subclasses	13
<b>1.9 Sobreposição</b>	<b>14</b>
1.9.1 Modificador final	15
1.9.2 @Override	15
1.9.3 Referência super	15
<b>1.10 Polimorfismo</b>	<b>16</b>
<b>1.11 Classes abstratas</b>	<b>16</b>
1.11.1 Métodos abstratos	17
<b>1.12 Interfaces</b>	<b>18</b>
1.12.1 Métodos default	18
<b>1.13 Coerção (casting) de referências</b>	<b>19</b>
1.13.1 instanceof	19

<b>2 Enumerações</b>	<b>20</b>
<b>3 Anotações</b>	<b>21</b>
<b>4 Classes internas</b>	<b>22</b>
4.1.1 Classes estáticas	22
4.1.2 Classes aninhadas (de instância)	22
4.1.3 Classes dentro de métodos	23
4.1.4 Classes anônimas	24
4.1.5 Interfaces funcionais e expressões lambda	24
<b>5 A classe java.lang.Object</b>	<b>25</b>
<b>5.1 Métodos de Object</b>	<b>25</b>
<b>5.2 Como estender Object</b>	<b>26</b>
5.2.1 <code>toString()</code>	26
5.2.2 <code>equals()</code>	26
5.2.3 <code>hashCode()</code>	27
5.2.4 <code>clone()</code>	27
<b>6 Classes parametrizadas</b>	<b>28</b>
<b>6.1 Comparable&lt;T&gt;</b>	<b>30</b>
<b>6.2 Comparator&lt;T&gt;</b>	<b>30</b>

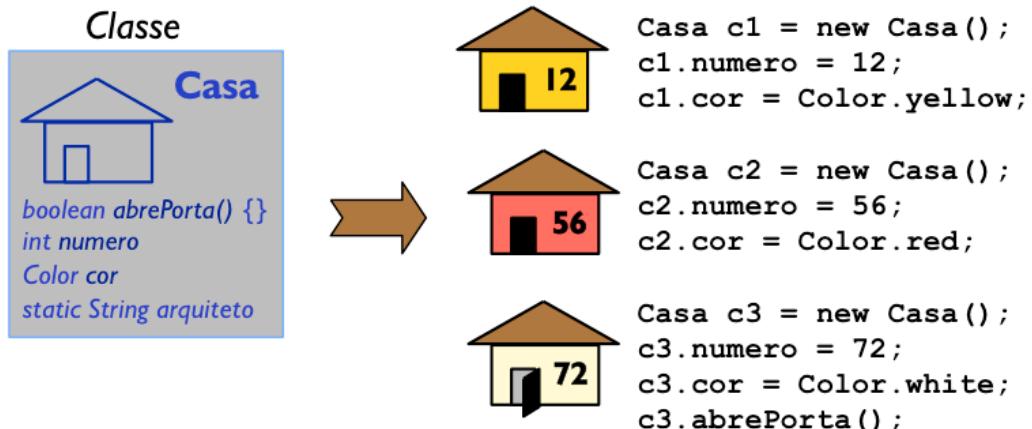
## 1 Classes

A base de qualquer aplicação Java é a *classe*. Uma classe é resultado de uma classificação abstrata. É uma abstração que representa um grupo de objetos e serve de *especificação* para criar um objeto.

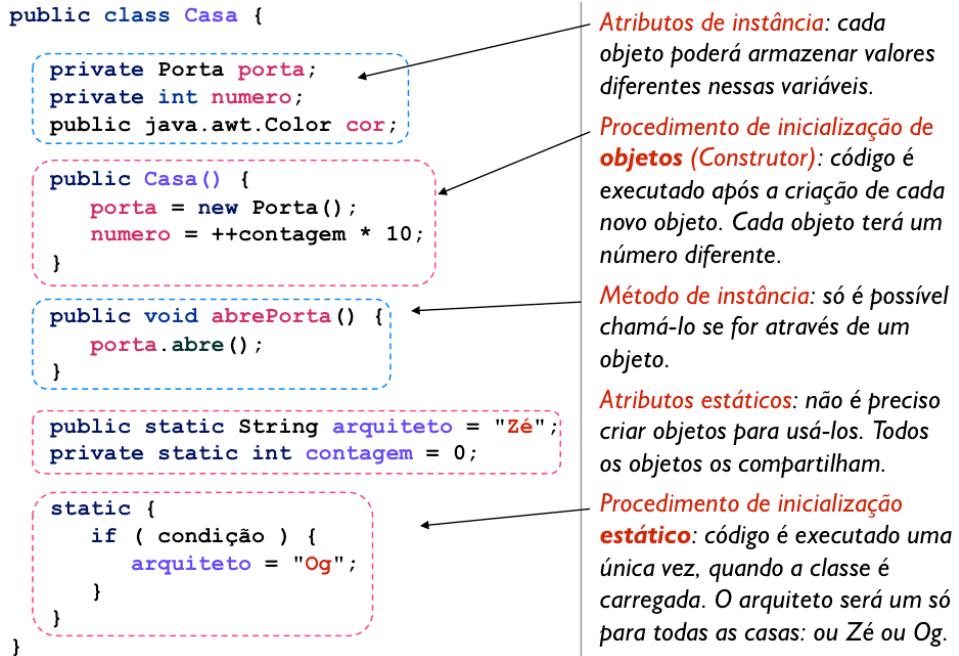
Uma classe também representa um *tipo de dados complexo*. Ela descreve a estrutura interna de um objeto (os tipos de dados dos atributos contidos nos objetos, os algoritmos de seus métodos) e sua *interface pública* (nomes de variáveis públicas e assinaturas de métodos públicos e construtores).

Uma analogia comum é que uma classe é uma planta (*blueprint*), ou projeto usado para construir *objetos*, as *instâncias* da classe:

*Instâncias da classe Casa (objetos)*



Uma classe é uma *estrutura de dados não ordenada*. A ordem das declarações de métodos, construtores, variáveis e outras estruturas não afeta seu funcionamento. É uma *convenção* e boa prática declarar variáveis e construtores no início da classe. Métodos e variáveis são chamados de *membros* de uma classe. Eles podem ser membros *estáticos* (se forem declarados com modificador *static*) ou de *instância*. Uma classe também pode conter procedimentos de instalação (construtores ou blocos *static*):



## 1.1 Membros de instância e membros estáticos

Antes de instanciar um objeto, a classe precisa ser carregada na memória. Isto é feito pelo *ClassLoader* quando a classe é usada pela primeira vez (ex: para declarar uma variável).

Os métodos e variáveis de *instância* de uma classe (todos os que não tiverem modificador *static*) só podem ser usados através de um objeto instanciado pela classe, mas métodos e atributos declarados com *static* já podem ser usados logo que a classe é carregada.

```
Casa c; // esta linha carrega a classe
```

A tentativa de usar um atributo ou método de instância através de uma variável declarada, mas que não referencia um objeto, provoca um erro de tempo de execução chamado de *NullPointerException*. Esta exceção informa que houve tentativa de acessar um ponteiro nulo.

```
c.abrePorta(); // causa NullPointerException, porque c não aponta para um objeto

int a = c.contagem // funciona, porque contagem é static (mas confunde)
int b = Casa.contagem; // forma recomendada de acessar membro estático

c = new Casa();
c.abrePorta(); // funciona: abrePorta() opera em objeto referenciado por c
```

## 1.2 Atributos (campos de dados)

Atributos são campos de dados dos objetos (ou classes, se tiverem modificador *static*). Também são chamados de *campos de dados*. A sintaxe mínima para declarar um atributo é:

```
tipo identificador;
```

Por exemplo:

```
Double preco;
```

Um *atributo* assim declarado será *uma variável de instância* (só pode ser usado através de um objeto instanciado) terá valor inicial *default* (0 ou null, dependendo do tipo), e será visível apenas dentro da classe e em classes do mesmo pacote.

### 1.2.1 Modificadores

Antes da declaração do *tipo* pode haver uma lista de *modificadores*, separados por espaços, em qualquer ordem. Modificadores são palavras que alteram aspectos do atributo, como acesso e qualidade. Abaixo uma lista dos modificadores que podem ser usados na declaração de atributos e seu significado:

- **final** – uma vez atribuído um valor a este atributo ele não pode ser alterado
- **private** – *modificador de acesso* – o atributo é visível apenas dentro da própria classe
- **public** – *modificador de acesso* – o atributo é visível em qualquer classe
- **protected** – *modificador de acesso* – o atributo é visível apenas dentro do pacote e em subclasses de outros pacotes
- **static** – é um atributo estático – só existe uma cópia e pode ser acessado através da classe, sem a necessidade de criar uma instância
- **transient** – usado em objetos serializáveis, indica que o atributo não faz parte do estado persistente do objeto (não é convertido em bytes quando o objeto é serializado)
- **volatile** – usado em aplicações com processamento paralelo, garante que o valor do campo estará sempre sincronizado (pode ser usado com segurança por threads que não sejam sincronizados).

Exemplos:

```
protected static final double PI = 3.14159;
transient InputStream entrada;
```

### 1.2.2 Valor inicial

Quando atributos são declarados em uma classe, eles podem receber um *valor inicial* através de atribuição:

```
private int x = 5;
private Ponto ponto = new Ponto(4,5);
int[] array = {1,2,3,4,5};
```

A inicialização é opcional para atributos estáticos ou de instância. Se um valor inicial não for atribuído, a variável será inicializada com um valor *default*, 0 ou null, dependendo do seu tipo.

A inicialização é obrigatória se a variável contiver os modificadores *static* e *final*, que representa uma constante cujo valor é imutável.

```
public static final GRAVIDADE = 9.8;
```

### 1.2.3 Anotações

Zero ou mais *anotações* podem aparecer antes das variáveis. As anotações não afetam as variáveis, mas servem de *meta-informação* para pré-processadores que interpretam o código-fonte. Algumas anotações são interpretadas pelo compilador.

```
@Id private int código;
```

## 1.3 Métodos

A sintaxe mínima para métodos em uma classe consiste do *tipo de retorno*, *nome* do método seguido por *parênteses* (obrigatórios, mesmo que não tenha argumentos) e um *bloco* entre chaves { ... } contendo as instruções Java que compõem a *implementação* do método. A declaração (que fica fora do bloco) é chamada de *assinatura do método*.

```
tipo-de-retorno nome() {}
```

Por exemplo:

```
int retornaNumero() {
    return 123;
}
```

A declaração pode ainda conter um ou mais *modificadores*, em qualquer ordem, antes do tipo de retorno, uma lista de *parâmetros* (declarações de variáveis locais entre parênteses, separadas por vírgula), e uma declaração *throws* após os parênteses contendo uma *lista de exceções* que *podem* ser causadas pelo método.

Anotações também podem aparecer antes dos modificadores e antes de cada parâmetro. Em *métodos genéricos*, um parâmetro de tipo (ex: <T>) pode aparecer antes do tipo de retorno. Se tiver um modificador *abstract* ou *native*, o corpo do método será *vazio* (a assinatura termina em ponto-e-vírgula)

### 1.3.1 Modificadores

Os modificadores que podem ser usados na declaração de um método estão listados abaixo:

- **public, private ou protected** – *modificadores de acesso* – limitam o acesso do método da mesma forma que os atributos. A ausência de um modificador de acesso restringe o acesso à própria classe e outras classes do mesmo pacote.
- **static** – é um método estático – só existe *uma cópia* e pode ser chamado através da classe sem que existam objetos instanciados; *não podem* acessar diretamente membros de instância da própria classe, sem que seja através de uma referência de objeto. Pode acessar diretamente apenas membros estáticos.
- **abstract** – é um método abstrato que define apenas a *interface* de um método cuja implementação será fornecida por uma subclasse; não contém corpo entre {...} – a declaração termina em ponto-e-vírgula.
- **final** – a implementação deste método não pode ser alterada por uma subclasse, através de sobreposição (*overriding*).
- **native** – define apenas a interface de um método cuja implementação é fornecida através de código em outra linguagem. Assim como métodos *abstract*, a declaração termina em ponto-e-vírgula.
- **strictfp** – estabelece que um método usará regras rígidas para operações em ponto flutuante (IEEE 754).
- **synchronized** – usado em aplicações que potencialmente realizam processamento paralelo, garante que a dois threads não possam modificar a instância ou classe ao mesmo tempo. Se for usada em um método *static*, o acesso à *classe inteira* (todas as instâncias) será restrita ao *thread* que estiver executando o método. Se usada em um método de instância, o *acesso ao objeto* será exclusivo do thread que executa o método.

Exemplos:

```
public static void main (String[] args ) { ... }
private native int metodo (int i, int j, int k);
final String abreArquivo() throws IOException, Excecao2 { ... }
```

### 1.3.2 Parâmetros

Parâmetros ou argumentos de um método são declarações de variáveis com escopo local ao método e deixam de existir quando o método termina. São separadas por vírgula:

```
public class Ponto {
    public double dist(Ponto outro) { ... }
    public double dist(int outroX, int outroY) { ... }
}
```

Na *chamada* de um método, os valores passados devem ser *compatíveis* com os tipos declarados, da mesma forma que uma operação de atribuição. A atribuição e a passagem de parâmetros são operações equivalentes:

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

    Ponto p1 = new Ponto(4,5);
    Ponto p2 = new Ponto(2,2);
    byte b = 20;

    double d1 = p1.dist( p2 );      // Equivale a: Ponto outro = p2;
    double d2 = p1.dist( b, 10);   // outroX = b, outroY = 10;
    double d3 = p1.dist( new Pixel(5, 5, "ff0000") ); // Ponto outro = new Pixel(...)

}
```

No exemplo acima, os comentários mostram as atribuições equivalentes, se aplicados na classe *Ponto* ilustrada anteriormente. Na segunda linha de chamada *dist()*, um *byte* foi convertido para *int*, para chamar o segundo método *dist()*. Na terceira chamada, um *Pixel* (subclasse de *Ponto*) foi chamada, fazendo com que uma referência do tipo *Ponto* (*outro*) aponte para uma instância do tipo *Pixel*. Essa referência terá acesso a todos os membros do *Pixel* que forem acessíveis através da interface herdada do *Ponto*, e apenas a esses membros

### 1.3.3 Varargs

Métodos podem receber parâmetros que são arrays. Para isto, há duas sintaxes. A primeira é simplesmente declarar que o método recebe um array:

```
public void método(int[] array) {
    int numero = array[1];
    ...
}
```

Outra forma é usar a notação de argumentos variáveis (*varargs*):

```
public void método(int... array) {
    int numero = array[1];
    ...
}
```

O resultado é o mesmo. Apenas um argumento com *varargs* pode existir em cada método. Se for usada em um método com vários argumentos, deve vir sempre no final, depois de todos os outros argumentos.

### 1.3.4 Variáveis locais

As variáveis declaradas dentro de parâmetros são locais e são inicializadas quando o método é chamado. Pode-se declarar variáveis locais em qualquer parte do método. Diferentemente das variáveis de instância, elas *não são automaticamente inicializadas* e precisam ser inicializadas explicitamente antes que sejam usadas. O compilador irá reclamar se houver a tentativa de usar uma variável local sem inicialização:

```
public void método() {
    int x;
```

```
if(x > 0) { ... } // erro de compilação: variável pode não ter sido inicializada!
...
}
```

Variáveis locais *podem* ter o mesmo nome que variáveis de instância ou estáticas. Não há conflito, mas a variável estática ou de instância *não será mais acessível* dentro do método. Ela é *ocultada* pela variável local. Embora permitido, usar nomes iguais para variáveis locais e de instância não é uma boa prática, exceto nos dois casos muito particulares de padrões consagrados: *métodos setter* e *construtores*, que distinguem a variável de instância através da referência this:

```
public class Ponto {
    private int x;
    public int getX(int x) {
        return x; // x é variável de instancia
    }

    public void setX(int x) {
        this.x = x; // x é variável local! this.x é variavel de instancia
    }
}
```

### 1.3.5 Exceções

As operações dentro de um método podem potencialmente causar exceções. Algumas exceções representam erros irrecuperáveis ou bugs no programa, mas outras, chamadas de *exceções checadas*, podem e devem ser tratadas. Exemplos são operações que interagem com recursos externos (rede, arquivos, bancos de dados), que são condições excepcionais que nada tem a ver com bugs ou problemas no ambiente de execução. É uma boa prática escrever código para lidar com essas condições excepcionais, oferecendo alternativas, ou pelo menos gravando logs dos problemas.

O tratamento de uma exceção *pode* ocorrer dentro de um método, mas é possível também que um método não possa *ou não deva* tentar lidar com uma exceção. Nesses casos, ele deixa que a exceção ocorra e retorna o controle para o thread que fez a chamada, declarando na sua assinatura, os tipos das exceções que podem ocorrer. Se um método potencialmente causa uma exceção checada, é obrigatório que ele ou capture a exceção, ou declare na assinatura do método.

Abaixo alguns exemplos de métodos declarando exceções:

```
public int read() throws IOException;
public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws IOException, ServletException { ... }
```

O mecanismo de exceções será explorado em maiores detalhes em outra seção.

### 1.3.6 Anotações

Métodos podem ter anotações declaradas em seus parâmetros ou antes dos modificadores. Anotações fornecem *meta-informação* para pré-processadores em determinados ambientes, e são, na sua maioria, ignoradas pelo compilador (mas elas precisam apenas ser importadas para que o código compile).

Uma anotação importante que é processada pelo compilador Java é a anotação `@Override`, que força o compilador a verificar se o método anotado realmente sobrepuja uma implementação:

```
@Override
public String toString() { ... }
@GET
@Type("text/plain")
public int getCodigo(@Param int a, @Param int b) { ... }
```

## 1.4 Construtores

Construtores não são métodos. São procedimentos especiais usados pelo operador `new` para instanciar objetos. A declaração de um construtor *parece* com a declaração de um método, já que tem parâmetros e um bloco de implementação, mas há algumas diferenças:

- Não têm tipo de retorno
- Podem ter apenas os modificadores `public`, `private` e `protected`
- Tem o mesmo nome que a classe.

Assim como métodos, construtores também podem declarar exceções.

A sintaxe mínima para um construtor é:

```
NomeDaClasse() {}
```

Este é o *construtor default*, que toda classe possui, mesmo que ele não seja declarado explicitamente. Portanto, declarar uma classe vazia:

```
public class UmaClasse {}
```

é equivalente a declarar uma classe contendo apenas o construtor default:

```
public class UmaClasse {
    public UmaClasse() {}
}
```

Observe que o construtor de uma classe pública (classe que podem ser acessada fora do próprio pacote) é também, por *default*, público.

Se um construtor for declarado sem o modificador `public`, em uma classe que tem esse modificador, a classe pode ser usada para *declarar* tipos e acessar métodos e atributos estáticos fora do pacote, mas objetos não poderão ser instanciados através do construtor.

## 1.5 Sobrecarga de nomes

Como construtores precisam ter o mesmo nome da classe, para que uma classe tenha várias opções de construtor, é preciso *repetir o nome*. Esse processo é chamado de *sobrecarga de nomes*. Como os nomes são iguais, o que distingue um construtor do outro é o *número ou os tipos* dos argumentos.

Mas deve-se ter cuidado para não abusar desse recurso. Existe a possibilidade de haver ambiguidade na hora de criar objetos. Na classe abaixo, os dois construtores se distinguem usando os critérios de tipo apenas:

```
public class Código {
    private long ativação;
    private int parcial;
    private long completo;

    public Código(int parcial, long ativação) {
        this.parcial = parcial;
        this.ativação = ativação;
        System.out.println("Ativação parcial");
    }
    public Código(long completo, long ativação) {
        this.completo = completo;
        this.ativação = ativação;
        System.out.println("Ativação completa");
    }
}
```

Não há risco de ambiguidade mas é difícil ter certeza qual construtor é chamado quando são passados valores inteiros que são convertidos:

```
public static void main(String[] args) {
    Código c1 = new Código(1,2); // int e int convertido para long - Primeiro
    Código c2 = new Código(1L, 2L); // long e long - Segundo
}
```

Mas se for acrescentado um terceiro construtor para Código, com (long,int), a primeira chamada causaria uma exceção, porque não seria possível descobrir qual construtor chamar. Ela teria que ser chamada com literais explícitos Código(1,2L).

A sobrecarga pode ser usada em métodos e construtores, embora as boas práticas recomendam que seu uso em métodos seja evitado, já que diminuem a clareza do código.

## 1.6 Blocos de inicialização

Usar métodos e variáveis estáticas em uma classe Java deve ser a exceção, e não a regra. O ideal é restringir o uso a casos comuns, descritos por padrões de design, como classes executáveis, singletons, utilitários, testes e outras situações em que seu uso é justificado.

Se uma classe que possui atributos estáticos precisar iniciá-los, isto pode ser feito através de um método, mas ele precisará ser chamado explicitamente. É possível inicializar atributos estáticos automaticamente através de um bloco *static*. A inicialização de blocos *static* é feita no momento em que a classe é carregada. Quaisquer exceções precisam ser tratadas dentro do bloco.

```
static List<Ponto> lista = new ArrayList<>();
static {
    lista.add(new Ponto(1,1));
    lista.add(new Ponto(2,2));
}
```

O uso de blocos static é raro. Mais raro ainda é o *inicializador de instância*, que consiste apenas do bloco {...} sem modificador static, que serve, como diz o nome, para inicializar o estado da instância (que pode ser feito no construtor).

## 1.7 Herança

Toda classe em Java herda métodos e atributos de uma outra classe. O nome dessa classe é conhecido na declaração da classe, pela sua cláusula *extends*:

```
public class NotaFiscal extends Documento {...}
```

Se uma classe não possui uma cláusula *extends*, ela herda da classe *java.lang.Object*. Portanto, todas as classes Java têm como ancestral a classe *java.lang.Object*. Ou seja:

```
class UmaClasse {}
```

é a mesma coisa que

```
class UmaClasse extends java.lang.Object {}
```

ou

```
class UmaClasse extends Object {}
```

já que todas as classes de *java.lang* são importadas automaticamente.

Isto significa que qualquer classe Java pode ter acesso aos métodos herdados por *Object* (a interface de herança de *Object* não disponibiliza variáveis). Podemos ver quais os métodos que existem em *Object* através de sua documentação (ou pelo Eclipse).

Dentre esses métodos estão *equals()*, que permite comparar um objeto com outro, *hashCode()*, que retorna um número para distinguir objetos em mapas, e *toString()* que gera uma representação *String* do objeto. Esses métodos são *métodos de instância*, o que significa que é necessário criar um objeto para ter acesso a eles. Portanto, podemos criar uma classe muito simples, como esta:

```
public class Minima {  
}  
}
```

Sem nenhum método, criar alguns objetos com ela:

```
public class MinimaTest {  
    public static void main(String[] args) {  
        Minima m1 = new Minima();  
        Minima m2 = new Minima();  
    }  
}
```

E chamar o método herdado de Object *toString()* para ver o resultado.

```
System.out.println(m1.toString());  
System.out.println(m2.toString());
```

O valor impresso corresponde ao nome completo da classe (pacote.Classe), um @ e o endereço de memória onde ele está armazenado. Como são dois objetos distintos, os endereços são diferentes.

Como *toString()* é um método conhecido, já que é de Object, vários métodos e operações em Java que trabalham com strings o chamam automaticamente. O exemplo mostrado funcionaria igual simplesmente passando o objeto para *println()*. Se um programa precisar de uma representação de qualquer objeto, ele chamará o método *toString()* dele:

```
System.out.println(m1);  
System.out.println(m2);
```

### 1.7.1 Como estender uma classe

Java suporta *herança simples*, o que significa que uma subclasse só pode ter uma superclasse. Algumas linguagens suportam *herança múltipla*. Portanto, a cláusula *extends* deve ser seguida pelo nome de uma única classe. Vamos estender a classe Ponto:

```
public class Ponto {  
    private int x;  
    private int y;  
  
    public Ponto() {}  
  
    public Ponto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
    ...  
}
```

Para criar a classe Pixel:

```
package br.com.agonavis.java.classes.geometria.bitmap;  
import br.com.agonavis.java.classes.geometria.Ponto;  
  
public class Pixel extends Ponto {}
```

A classe Pixel agora tem acesso a *todos os métodos* de Ponto. Mas ela já tinha acesso a eles sem estender. A classe Linha, por exemplo, consegue chamar os métodos getX() e getY():

```
public class Linha {
    private Ponto a, b;
    ...

    public double comprimento() {
        long q = (a.getX() - b.getX()) * (a.getX() - b.getX())
            + (a.getY() - b.getY()) * (a.getY() - b.getY());
        return Math.sqrt(q);
    }
}
```

A diferença não está no *acesso* a esses dados, mas no que a herança *representa*. O Pixel agora é considerado *um tipo de Ponto*. Pode-se dizer que o Pixel é *um Ponto*. O Pixel não precisa declarar nenhum método novo, e outra classe pode, através de uma instância de Pixel, obter seus valores de x e y.

```
import br.com.argonavis.java.classes.geometria.bitmap.Pixel;

public class PixelTest {
    public static void main(String[] args) {
        Pixel pixel = new Pixel();
        System.out.println("x: " + pixel.getX());
        System.out.println("y: " + pixel.getY());
    }
}
```

O Pixel até aqui *herda* de Ponto, mas não acrescentou nada para estendê-lo. Um Pixel é um Ponto com informação de cor, portanto podemos acrescentar um atributo int que guarde o valor de uma cor, e métodos getter e setter:

```
public class Pixel extends Ponto {
    private int cor = 0xffffffff; // cor default

    public int getCor() {
        return cor;
    }

    public void setCor(int corNova) {
        cor = corNova;
    }
}
```

## 1.7.2 Construtores, super() e this()

Se a classe Ponto *não tivesse* o construtor default Ponto(), mas apenas o construtor que recebe x e y, teríamos um erro de compilação na hora de criar o Pixel:

```
Implicit super constructor Ponto() is undefined for default constructor. Must define an
explicit constructor      Pixel.java      line 5
```

Isto porque a construção de um objeto é um processo de várias etapas, que passa pela inicialização dos atributos e execução dos construtores em todas as superclasses até Object. A operação

```
Pixel px = new Pixel();
```

Causa as seguintes etapas:

1. Pixel: Inicialização de cor para valor 0
2. Ponto: Inicialização de x e y para valores zero
3. Object: Execução do construtor Object()
4. Ponto: Inicialização de x e y para valores explícitos (se houver)

5. Ponto: Execução do construtor Ponto()
6. Pixel: Inicialização de cor para valores explícitos: 0xffffffff
7. Pixel: Execução do construtor Pixel()

Veja o que acontece no passo número 5. O construtor default da superclasse (Ponto(), sem argumentos) é chamado. Como Ponto não possui um construtor default, ocorre um erro na criação do Pixel.

Pixel também não tem um construtor Pixel(), mas como Pixel não tem nenhum outro construtor declarado explicitamente, ele possui um que é implícito.

Há duas formas de resolver o problema.

- Se tivermos acesso à classe Ponto, podemos criar um construtor default para ele.
- Se não temos acesso ao ponto, precisamos substituir a chamada do construtor default, por uma chamada ao construtor de Ponto que recebe dois ints. Isto é feito com a instrução super().

O construtor de Pixel, implícito, tem esta estrutura:

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

Ou seja, a chamada a super() também é implícita. Qualquer construtor explícito que não declarar explicitamente uma instrução super(), também tem uma chamada implícita a super() na sua primeira linha. Para selecionar o construtor de dois argumentos, podemos declarar explicitamente o construtor de Pixel e chamar super() com esses dois argumentos:

```
public Pixel() {
    super(0,0); // chama o constructor de dois ints da superclasse: Ponto(int,int)
}
```

Se houver super() em um construtor, deve aparecer sempre na primeira linha.

Similar a super é *this()*. Não confunda com a referência *this*, que é uma referência ao próprio objeto. *this()* é uma chamada a um construtor da própria classe. Assim como super(), o construtor é identificado pela quantidade de argumentos. Assim *this(0,0)* chama o construtor da própria classe que recebe dois inteiros. Usando *this()* podemos evitar repetir código dentro de construtores, redirecionando todas as chamadas para um único construtor:

```
public class Pixel extends Ponto {
    private int cor;

    public Pixel() {
        this(0,0, 0xff0000); // chama o constructor de tres ints desta classe!
    }

    public Pixel(int x, int y, int cor) {
        super(0,0); // chama o constructor de dois ints da superclasse
        this.cor = cor;
    }
    ...
}
```

### 1.7.3 Classes finais

Pode-se impedir que uma classe seja estendida declarando-a como *final*.

```
public final class Definitiva { ... }
```

A tentativa de incluir uma classe final em uma cláusula extends causa um erro de compilação.

## 1.8 Encapsulamento e modificadores de acesso

O encapsulamento de uma classe ou objeto estabelece uma *interface* que determina como essa classe pode ser usada. Nos exemplos acima usamos *private* para limitar o acesso de atributos apenas dentro da classe, e declaramos os métodos e construtores como *public*. Portanto, a *interface pública* consiste apenas dos métodos e construtores da classe. Essa interface pública serve para tanto para *usar* uma classe (instanciar objetos, chamar métodos) como para *estendê-la* através de uma subclasse.

Usando os modificadores de acesso do Java podemos estabelecer outras interfaces, mais abrangentes e mais restritivas, para controlar melhor como uma classe é usada.

### 1.8.1 Interface privativa ao pacote

Para outras classes dentro do mesmo pacote, a interface pública de uma classe pode ser maior. Quaisquer membros que *não tenham nenhum modificador de acesso* são acessíveis dentro do pacote como se estivessem declarados com *public*. Mas, para classes de outros pacotes, esses membros são inacessíveis, como se tivessem sido declaradas com *private*.

Por exemplo, esta versão da classe Ponto não declara modificador de acesso nos seus atributos x e y:

```
public class Ponto {
    int x;
    int y;
    ...
}
```

Portanto a classe *Linha*, que *pertence ao mesmo pacote* que *Ponto*, pode acessá-los diretamente, como se fossem *public*:

```
public class Linha {
    private Ponto a, b;
    ...

    public double comprimento() {
        long q = (a.x - b.y) * (a.y - b.y)
            + (a.y - b.y) * (a.y - b.y);
        return Math.sqrt(q);
    }
}
```

Mas uma classe que estiver em outro pacote só poderá usar a interface pública, que não dá acesso a atributos:

```
public class LinhaDePixels {
    private Pixel a, b;
    ...

    public double comprimento() {
        long q = (a.getX() - b.getX()) * (a.getX() - b.getX())
            + (a.getY() - b.getY()) * (a.getY() - b.getY());
        return Math.sqrt(q);
    }
}
```

### 1.8.2 Interface para subclasses

A classe *Pixel* está em um pacote diferente de *Ponto* e *Linha*, portanto não tem acesso aos atributos de *Ponto*. Mas se declararmos esses atributos como *protected*, eles farão parte da interface de extensão da classe, e serão acessíveis por subclasses:

```

package geometria;

public class Ponto {
    protected int x;
    protected int y;
    ...
}

package geometria.bitmap;
import geometria.Ponto;
public class Pixel extends Ponto {

    ...
    public double distancia(Pixel p) {
        long q = (x - p.x) * (x - p.x) + (y - p.y) * (y - p.y);
        return Math.sqrt(q);
    }
}

```

Membros *protected* são acessíveis por qualquer classe *dentro* do pacote, mas também por subclasses de fora do pacote.

Apesar do Pixel ter herdado as variáveis x e y via *protected*, elas são acessíveis apenas para a classe Pixel, e não para outras classes do mesmo pacote que Pixel, mesmo que chamadas através da classe Pixel. Por exemplo, a classe *LinhaDePixels*, que está no mesmo pacote de Pixel não pode acessar as variáveis *protected* x e y que Pixel herdou. O trecho abaixo provoca erro de compilação:

```

package geometria.bitmap;
public class LinhaDePixels {
    private Pixel a, b;
    ...
    public double comprimento() { // CAUSA ERRO DE COMPILAÇÃO!
        long q = (a.x - b.y) * (a.y - b.y)
            + (a.y - b.y) * (a.y - b.y);
        return Math.sqrt(q);
    }
}

```

## 1.9 Sobreposição

Uma subclasse pode estender outra com novos métodos. Além disso, ela também pode *redefinir* a implementação dos métodos herdados. Isto é chamado de *sobreposição*, *sobreescrita*, *redefinição* ou *overriding*.

Para sobrepor um método, a subclasse deve declarar uma interface para o método com uma *assinatura compatível* (mesmo nome, tipo de retorno igual ou derivado, quantidade e tipo de parâmetros, lista de exceções igual ou menos abrangente).

Se o método original tiver um modificador de acesso, o método sobreposto pode ter um modificador igual ou *mais abrangente*. Ou seja, se o original for *public*, o sobreposto só pode ser *public*. Se o original for *protected*, o sobreposto pode ser *protected*, mas também pode ser *public*.

O tipo de retorno não precisa ser igual, mas precisa ser um *subtipo*. Se for um tipo primitivo, tem que ser um tipo compatível (que caiba no tipo de retorno do método redefinido). Se for uma classe, deve ser uma subclasse.

Se apenas os parâmetros tiverem um tipo diferente, não ocorrerá uma sobreposição, mas uma sobrecarga, s ambas os métodos permanecerão disponíveis na subclasse (a sobreposição mudando os tipos dos argumentos pode ser feita apenas em métodos cujos argumentos são parâmetros de classe.)

A classe Ponto possui um método imprimir, que retorna um string com seu valor x e y:

```
public String imprimir() {
    return "("+x+","+y+");
```

Como a classe Pixel tem um atributo a mais, deve redefinir o método imprimir() para que ele acrescente esse atributo:

```
public class Pixel extends Ponto { ...
    public String imprimir() {
        return "("+x+","+y+"), cor: " + Integer.toHexString(cor);
    }
}
```

Uma vez sobreposto, a chamada do método usando uma instância da subclasse usará *sempre* a implementação da subclasse:

```
Pixel pixel = new Pixel();
Ponto ponto = pixel;
System.out.println( ponto.imprimir() ); // chama imprimir() de Pixel!
```

Imprime:

(0,0), cor: ff0000

### 1.9.1 Modificador final

Pode-se impedir que uma subclasse sobrescreva um método declarando-o como private, mas assim ela também não poderá executá-lo. Se a intenção for permitir o acesso, mas impedir a alteração, o método deve ser declarado com *final*. Métodos declarados com esse modificador não podem ser redefinidos.

```
public final void computarSenha() { ... }
```

### 1.9.2 @Override

Se um método com mesmo nome em uma subclasse não tiver o mesmo tipo e quantidade de argumentos de um método herdado de uma superclasse, não irá ocorrer uma redefinição, mas uma sobrecarga. Ou seja, a subclasse terá dois métodos com o mesmo nome, que se distinguem pelo número e tipo de argumentos. Isto pode ocorrer accidentalmente. Para evitar esse tipo de erro e forçar o compilador a verificar a sobreposição, deve-se usar a anotação *@Override* antes da declaração do método:

```
public class Pixel extends Ponto {
    private int cor;
    ...
    @Override
    public String imprimir() {
        return "("+x+","+y+"), cor: " + Integer.toHexString(cor);
    }
}
```

O compilador então verificará se existe algum método, dentre os métodos herdados das subclasses, que tenha uma assinatura compatível para realizar a redefinição. Se não houver, haverá um erro de compilação.

### 1.9.3 Referência super

Além do super(), usado em construtores, existe também uma *referência super*. Vimos antes que *this* é uma referência para o próprio objeto e *this.x* pode ser usado para referenciar um membro do objeto. De forma análoga, *super* pode ser usada para acessar os valores originais de atributos que foram redefinidos, ou métodos que foram sobrepostos. O uso mais comum é em métodos. Por exemplo, para imprimir um pixel, podemos imprimir as coordenadas do Ponto (mesmo

conteúdo que o método `imprimir()` do *Ponto*), e *acrescentar* as informações de cor. Isto pode ser feito com `super` da seguinte maneira:

```
@Override
public String imprimir() {
    return super.imprimir() + ", cor: " + Integer.toHexString(cor);
}
```

Embora seja permitido, não é uma boa prática usar `super` para referenciar um método diferente daquele que está sendo sobreposto.

## 1.10 Polimorfismo

Quando um método é chamado usando uma referência do tipo *Ponto*, e o resultado da execução ocorre em uma instância *Pixel*, está havendo Polimorfismo. Por exemplo, este trecho de código mostrado anteriormente é Polimorfismo:

```
Pixel pixel = new Pixel();
Ponto ponto = pixel;
System.out.println( ponto.imprimir() ); // chama imprimir() de Pixel!
```

Polimorfismo significa muitas formas. O texto `ponto.imprimir()` não garante que está operando em um *Ponto*. Pode ser qualquer subclasse. Outro exemplo mostrado de polimorfismo está em:

```
Minima m2 = new Minima();
System.out.println(m1); // chama toString() implicitamente
```

O método `println()` recebe como argumento um *Object*, portanto pode receber uma instância de qualquer classe do Java. A única coisa que ele precisa fazer é chamar `toString()`, e será impresso ou o valor original, caso nenhuma das superclasses da instância tenha redefinido `toString()`, ou o valor redefinido pela própria classe ou pela última superclasse que redefiniu o método.

Polimorfismo é um recurso muito poderoso e tem importância central em aplicações orientadas a objetos. Usando polimorfismo é possível reduzir a quantidade de estruturas condicionais em código procedural, encapsulando código em objetos que são selecionados com base no seu tipo. Vários padrões de design são construídos sobre o conceito de polimorfismo e Java oferece muitos recursos para implementá-lo de forma eficiente através de classes abstratas e interfaces.

## 1.11 Classes abstratas

Uma classe às vezes representa um conceito abstrato, que não pode ser completamente especificado. Por exemplo, a classe *Figura*

```
package br.com.agonavis.java.classes.geometria;

public class Figura {}
```

O que é uma *Figura*? Com que ela se parece?

Uma *Figura* é algo que possui uma *área*. Pode ser, por exemplo, um *Retângulo* ou um *Círculo*, mas a sua forma, quando chamada simplesmente de “*Figura*”, não é especificada. Em Java, um conceito abstrato pode ser representado por uma classe abstrata, que deve ser declarada com o modificador `abstract`:

```
public abstract class Figura {
```

Uma classe declarada como `abstract` pode ser usada para declarar tipos, mas não pode ser usada para construir objetos com `new`. Para isto, é preciso criar uma subclasse concreta. Por exemplo, se *Retangulo* e *Circulo* forem subclasses de *Figura*:

```
class Retangulo extends Figura {}
class Circulo extends Figura {}
```

Podemos declarar variáveis do tipo abstrato Figura:

```
Figura fig1, fig2;
```

Que irão receber depois referências para instâncias de classes concretas:

```
fig1 = new Circulo(50);
fig2 = new Retangulo(30,40);
```

O método area() de Figura pode ser chamado, mas ele não faz sentido nessas subclasses:

```
public abstract class Figura {
    public double area() {
        return 0; // não sabemos qual a área de uma Figura abstrata
    }
}
```

Sabemos como calcular as áreas de um Circulo ou de um Retangulo. Portanto, precisamos redefinir o método area() em cada subclasse, já área de um Circulo é diferente da área de um Retangulo:

```
public class Circulo extends Figura {
    private int raio;

    public Circulo(int raio) {
        this.raio = raio;
    }

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

public class Retangulo extends Figura {
    private int altura, largura;

    public Retangulo(int altura, int largura) {
        this.altura = altura;
        this.largura = largura;
    }

    @Override
    public double area() {
        return altura * largura;
    }
}
```

Agora é possível chamar, de forma polimórfica, os métodos area() de fig1 e fig2 e obter as áreas corretas de cada figura.

```
System.out.println("Área da Figura 1: " + fig1.area());
System.out.println("Área da Figura 2: " + fig2.area());
```

```
Área da Figura 1: 7853.981633974483
Área da Figura 2: 1200.0
```

Mas se *esquecermos* de redefinir a area() em alguma subclasse, ela irá retornar um valor incorreto.

### 1.11.1 Métodos abstratos

O método area() em Figura é um conceito abstrato, que não pode ser especificado na classe Figura, mas *deve* ser especificado nas subclasses. Essa obrigação pode ser implementada em Java

declarando o método sem corpo, com a assinatura terminando em ponto-e-vírgula, e com o modificador *abstract*:

```
public abstract class Figura {
    public abstract double area();
}
```

Com essa declaração, uma subclasse não será considerada uma classe concreta enquanto não redefinir o método contendo uma implementação concreta (mesmo que vazia). Não é mais possível esquecer de implementar o método nas subclasses. O compilador sempre irá reclamar se uma subclasse que estende uma classe abstrata não implementa *todos* os seus métodos abstratos. Se ela não o fizer, ela precisa *também* ser declarada como *abstract*.

## 1.12 Interfaces

Se todos os métodos de uma classe abstrata forem abstratos, ela pode ser definida como uma *interface*. Uma interface geralmente é usada para isolar um conceito abstrato que pode ser reusado. Por exemplo, a possibilidade de calcular a área poderia ser usado por outras classes e isolado em uma interface Superficie:

```
public interface Superficie {
    double area();
}
```

Em uma interface, todos os métodos são *public* e *abstract*, portanto não é preciso incluir esses modificadores. Agora podemos redefinir a classe Figura como:

```
abstract class Figura implements Superficie { }
```

E tudo funcionará como antes. Embora a classe *declare* implementar a interface, ela não o implementa de verdade (por isso precisa ser declarada como *abstract*).

### 1.12.1 Métodos default

Um problema das interfaces é se for necessário acrescentar um método abstrato novo, haverá efeitos colaterais em todas as subclasses, que precisarão também implementar esse novo método.

Uma solução para isto existe a partir do Java 8 usando *métodos default*. Uma interface pode incluir uma implementação concreta declarando-o como *default*:

```
default double perimetro() {
    return 0;
}
```

Um método *default* pode ainda ser sobreposto numa subclasse (mas não precisa), e pode também ser redeclarado para se tornar um método abstrato.

Em alguns casos, raros, pode haver conflitos. Neste caso é preciso implementar a interface novamente. Por exemplo se as duas interfaces *Superficie* e *Contornavel* tiverem um método *default double perimetro()*, a subclasse precisa sobrepor e escolher uma implementação (ou criar outra):

```
public class Figura implements Superficie, Contornavel {
    @Override
    public double perimetro() {
        return Superficie.super.perimetro(); // escolhe implementação de Superficie
    }
}
```

## 1.13 Coerção (casting) de referências

Uma superclasse sempre pode receber uma referência de uma subclasse:

```
Object obj = new Retangulo(50,50);
```

Isto é possível porque todos os membros que existem na *interface* proporcionada pela superclasse existem na subclasse. Ou seja, no exemplo acima, será possível chamar na instância de Retangulo:

```
String s = obj.toString();
boolean b = obj.equals(new Retangulo(50,50));
```

Mas, se a subclasse definir novos métodos, eles não podem ser chamados *através dessa referência*, porque ela foi declarada com a interface da superclasse. Isto é comum quando um método que recebe uma superclasse, recebe uma instância de uma subclasse:

```
public boolean equals(Object obj) { ... }
```

Mas poderá ser necessário, dentro do método, chamar um método que existe no objeto, mas que não é acessível através da superclasse de sua referência. Por exemplo, como chamar a *area()* ou o *perimetro()* dentro de *equals(Object)*? Uma solução seria *copiar a referência* para uma outra variável, declarada com o tipo da subclasse, que tem uma interface mais abrangente:

```
public boolean equals(Object obj) {
    Retangulo novaRef = obj; //NÃO COMPILA!
    double area = novaRef.area();
    ...
}
```

Mas isto não compila. Por que? O *compilador* não permite que uma superclasse seja convertida em uma subclasse, mesmo que o objeto referenciado seja do tipo compatível. Assim como ocorre com tipos primitivos, é necessário, nesses casos, fazer uma coerção usando o operador de cast:

```
public boolean equals(Object obj) {
    Retangulo novaRef = (Retangulo)obj; //AGORA COMPILA!
    double area = novaRef.area(); // Isto funciona se obj for mesmo um Retangulo
    ...
}
```

Agora, é possível chamar o método *area()*, que só existe em Retangulo. Não haverá erro a menos que o objeto recebido não seja um Retangulo, como esperado.

### 1.13.1 Instanceof

Para evitar a dúvida acima, antes de fazer um cast, é preciso ter certeza que a referência contida em uma variável realmente aponta para uma instância da classe para a qual se deseja converter. Isto pode ser feito com a operação *instanceof*, que permite testar se uma referência é uma instância de uma classe.

```
if(obj instanceof Retangulo) { ... }
```

O teste com *instanceof* retorna *true* com qualquer superclasse da classe que instanciou o objeto referenciado, até mesmo Object. É suficiente para garantir que o método mostrado anteriormente não tenha erros de cast:

```
public boolean equals(Object obj) {
    if(obj instanceof Retangulo) {
        Retangulo novaRef = (Retangulo)obj;
        double area = novaRef.area(); // Funciona pois obj é Retangulo!
        ...
    }
}
```

Se por algum motivo for necessário saber exatamente a classe usada para construir a instância do objeto, é preciso comparar usando o método `getClass()`:

```
if(obj.getClass().equals(Retangulo.class)) {  
}
```

## 2 Enumerações

Enumerações são coleções de constantes, geralmente usadas em expressões condicionais. Uma enumeração é criada em Java usando a palavra `enum`:

```
public enum Estado {  
    CONECTADO, CONECTANDO, DESLIGADO, DESCONECTANDO;  
}
```

E podem ser usados em métodos e expressões:

```
public class Modem {  
    void conectar(Estado status) {  
        switch(status) {  
            case Estado.CONECTADO:  
                break;  
            case Estado.DESCONECTADO:  
                iniciarConexao();  
                ...  
        }  
        ...  
    }  
}
```

Enums podem ser declarados em unidades de compilação independentes, mas é comum usá-los também como membros de instância das classes que os utilizam:

```
public class Card {  
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,  
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }  
  
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }  
  
    private static final List<Card> protoDeck = new ArrayList<Card>();  
  
    private final Rank rank;  
    private final Suit suit;  
  
    private Card(Rank rank, Suit suit) {  
        this.rank = rank;  
        this.suit = suit;  
    }  
  
    public String toString() { return rank + " of " + suit; }  
  
    static {  
        for (Suit suit : Suit.values())  
            for (Rank rank : Rank.values())  
                protoDeck.add(new Card(rank, suit));  
    }  
}
```

Enumerações podem ter métodos e atributos, que podem ser usados para obter mais informações associados à constante.

```
public enum Planet {  
    VENUS (4.869e+24, 6.0518e6),  
    EARTH (5.976e+24, 6.37814e6),  
    MARS (6.421e+23, 3.3972e6);
```

```

private final double mass;    // in kilograms
private final double radius; // in meters
Planet(double mass, double radius) {
    this.mass = mass;
    this.radius = radius;
}
private double mass() { return mass; }
private double radius() { return radius; }
// universal gravitational constant (m3 kg-1 s-2)
public static final double G = 6.67300E-11;
double gravity() {
    return G * mass / (radius * radius);
}
double weight(double otherMass) {
    return otherMass * surfaceGravity();
}
}
}

```

### 3 Anotações

Anotações são declaradas como interfaces. Para criar uma anotação que será usada em tempo de execução para anotar classes, é preciso declarar uma `@interface` da forma:

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CLASS) // pode ser usada antes de classes
public @interface Anotacao {}

```

Uma vez criada, ela pode ser usada para anotar classes:

```

@Anotacao
public class UmaClasse {}

```

Anotações podem ter atributos. Eles são declarados como métodos, com um tipo e valor *default*. Tipicamente contém valores simples como strings, números e enums, mas podem conter também coleções de outras anotações. A seguir um exemplo com dois atributos:

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CLASS) // pode ser usada antes de classes
public @interface Anotacao {
    public enum Tamanho { PEQUENO, MEDIO, GRANDE }
    String value() default "";
    Tamanho tamanho() default Tamanho.GRANDE;
}

```

Que pode ser usada sem argumentos (já que todos são default), com apenas um ou com os dois:

```

@Anotacao(value="Teste", tamanho=Tamanho.MEDIO)
public UmaClasse { ... }

```

A recuperação de anotações e seus atributos é realizado usando a API de reflexão, que permite acesso à classe e seus métodos. Anotações podem ser lidas em tempo de execução se forem criadas com declarando isto através de `@Retention(RetentionPolicy.RUNTIME)`.

É preciso saber onde a anotação foi definida. Se for em uma classe, pode-se usar o método `getAnnotation()` diretamente no objeto que representa a classe:

```
UmaClasse.class.getAnnotation(Anotacao.class);
```

ou

```
UmaClasse c = new UmaClasse ();
c.getClass().getAnnotation(Anotacao.class);
```

Se for em um método, atributo, parâmetro, é preciso usar outros métodos da API de reflection para primeiro obter uma instância que represente esse elemento, e depois chamar `getAnnotation()`.

## 4 Classes internas

Classes, enums e interfaces normalmente são elementos top-level e ocorrem na raiz de uma unidade de compilação. Mas é possível também declarar esses elementos dentro de outras classes e até mesmo dentro de métodos.

Várias classes ou interfaces podem ser declaradas dentro de uma classe ou interface. Elas tornam-se membros da classe externa. Podem ter modificadores de acesso `public`, `private` e `protected`, e podem ser `static` ou não.

### 4.1.1 Classes estáticas

Classes `static` funcionam da mesma forma que classes externas, apenas ganham o nome da classe externa como prefixo. No exemplo abaixo temos uma classe `InnerClassTest1` que contém uma declaração estática de interface (`Interface`) e duas classes (`Inner1` e `Inner2`):

```
public class InnerClassTest1 {

    public static interface Interface {
        void m1();
    }

    private static class Inner1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Hello 1");
        }
    }

    private static class Inner2 implements InnerClassTest1.Interface {
        private String nome;
        public Inner2(String nome) {
            this.nome = nome;
        }
        public void m1() {
            System.out.println("Hello 2: " + nome);
        }
    }
}
```

O nome de cada uma dessas classes, dentro do seu pacote, contém o nome da classe eterna como prefixo. Veja a cláusula `implements` de `Inner2`, que implementa a interface `Interface`.

### 4.1.2 Classes aninhadas (de instância)

Se uma classe for declarada dentro de outra e *não tiver* modificador `static`, ela é uma *classe aninhada* que só existirá quando uma instância da classe externa for criada.

```
public class InnerClassTest2 {
    private String nome = "Outer World";

    private class Inner2 implements InnerClassTest1.Interface {
        public String nome;

        public Inner2(String nome) {
            this.nome = nome;
        }
    }
}
```

```

        public void m1() {
            System.out.println("Hello 2: " + nome);
        }
    }

    public void outerMethod() {
        Inner2 i2 = new Inner2("Inner World");
        ...
    }
}

```

Portanto, para criar uma instância da classe interna, ou mesmo para acessar seus campos externos, tem que haver primeiro uma instância da classe externa. Ela pode ser chamada normalmente dentro de um método de instância da classe externa (como mostrado em outerMethod) usando a referência corrente. A instrução acima também pode ser escrita da seguinte forma:

```

public void outerMethod() {
    Inner2 i2 = this.new Inner2("Inner World");
    ...
}

```

Em outra classe ou em um método estático, não há this e é preciso prefixar o new com a instância dentro da qual a classe está sendo instanciada:

```

InnerClassTest2 outer = new InnerClassTest2();
InnerClassTest2.Inner2 i2 = outer.new Inner2("Inner World");

```

Uma classe aninhada também possui acesso a diferentes referências this. Dentro do método m1, a referência this sempre refere-se à classe interna. Para referenciar um membro da classe externa, é preciso prefixar o this com o nome da classe externa. O método m1() abaixo mostra todas as opções:

```

public class InnerClassTest2 {
    private String nome = "Outer World"; // InnerClassTest2.this.nome

    private class Inner2 implements InnerClassTest1.Interface {
        public String nome; // this.nome
        public Inner2(String nome) {
            this.nome = nome;
        }
        public void m1() {
            System.out.println("Hello 2: " + nome);
            System.out.println("Hello 2: " + this.nome);
            System.out.println("Hello 2: " + InnerClassTest2.this.nome);
        }
    }
    ...
}

```

### 4.1.3 Classes dentro de métodos

Classes também podem ser declaradas dentro de métodos. São classes criadas para serem instanciadas, usadas e descartadas depois que o método terminar. Geralmente elas são criadas e instanciadas imediatamente. Como o método tem uma estrutura sequencial, a classe precisa ser declarada antes de ser instanciada:

```

import br.com.agonavis.java.classes.FunctionalInterfaceTest.Interface;

public class InnerClassTest3 {

    public static void method1(ActionListener s) {
        s.actionPerformed(null);
    }
}

```

```

public static void method2(Interface i) {
    i.m1();
}

public static void main( String[] args ) {
    System.out.println( "Hello World!" );

    class Inner1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            System.out.println("Hello 1");
        }
    }
    class Inner2 implements Interface {
        public void m1() {
            System.out.println("Hello 2");
        }
    }

    ActionListener i1 = new Inner1();
    Interface i2 = new Inner2();

    method1(i1);
    method2(i2);

}
}

```

A classe deixa de existir depois que o método termina.

No exemplo acima, duas classes são definidas dentro de um método. Ambas implementam uma interface contendo um método. As duas são posteriormente instanciadas e passadas para métodos que foram declarados na classe externa.

#### 4.1.4 Classes anônimas

Existe uma sintaxe compacta e alternativa a classes usadas dentro de métodos que não precisa de nome. Essas classes geralmente são criadas para implementar alguma interface e executar um único método. Essa sintaxe é muito comum no tratamento de eventos.

As classes internas mostradas no exemplo anterior podem ser transformadas em classes anônimas, declaradas e instanciadas imediatamente sem a necessidade de uma classe extra. Todo o código em destaque na listagem anterior pode ser substituído pelo código abaixo:

```

method1( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hello 1");
    }
} );

method2( new Interface() {
    public void m1() {
        System.out.println("Hello 2");
    }
} );

```

Observe que new é usado para chamar o construtor da interface, e que a classe que implementa a interface na verdade não tem nome.

#### 4.1.5 Interfaces funcionais e expressões lambda

Interfaces funcionais são interfaces que contém um único método abstrato. Elas têm um significado especial em Java 8. As interfaces que usamos acima, nas classes anônimas são interfaces funcionais.

Não é necessário, mas é uma boa prática usar a anotação `@FunctionalInterface` antes de uma interface funcional, para que o compilador reclame caso a interface anotada não seja de fato uma interface funcional.

As expressões mostradas acima podem ser expressas usando lambda de maneira ainda mais compacta:

```
method1( ActionEvent e ) -> System.out.println("Hello 1") ;
method2( () -> System.out.println("Hello 2") );
```

Desaparecem todas as chaves, e a interface é identificada pelos parâmetros do método que contém.

## 5 A classe `java.lang.Object`

A classe `java.lang.Object` é herdada por todas as classes Java. É muito importante conhecer os métodos de `Object` pois eles têm um papel importante em várias estruturas essenciais da linguagem, como cópia, impressão, comparação, ordenação e coleção de objetos. Vários desses precisam ser redefinidos nas subclasses

### 5.1 Métodos de Object

A classe `Object` contém 11 métodos. Na prática são 8, pois 3 são versões sobrecarregadas ou com funcionalidades similares. As assinaturas dos métodos estão listadas abaixo.

<code>boolean equals(Object o)</code>	Deve retornar true para objetos considerados iguais, e false se objetos não forem iguais. O critério de igualdade deve ser determinado pela subclasse. Dois objetos considerados iguais devem retornar o mesmo valor para <code>hashCode()</code> . A implementação default retorna true se os dois objetos tiverem a mesma referência (se forem o mesmo objeto).
<code>int hashCode()</code>	Retorna um número que será usado em algoritmos de <code>hash</code> para classificar objetos. Deve ser sobreposto sempre que <code>equals()</code> for sobreposto para que dois objetos iguais sempre tenham o mesmo <code>hashCode</code> . Dois objetos diferentes <i>podem</i> ter o mesmo <code>hashCode</code> . A implementação default retorna um número correspondente ao endereço do objeto no heap.
<code>String toString()</code>	Retorna uma representação do objeto em forma de <code>String</code> . Deve ser sobreposto em cada classe para fornecer uma representação própria. A implementação original retorna o nome da classe @ e o <code>hashCode()</code> em hexadecimal
<code>protected Object clone()</code>	Cria e retorna uma cópia rasa do objeto (cópia apenas valores primitivos e referências). Deve ser redeclarado em subclasses clonáveis que implementam a interface <code>Cloneable</code> e capturar a <code>CloneNotSupportedException</code> .

<code>Class&lt;?&gt; getClass()</code>	Retorna um objeto do tipo Class, que representa a classe deste objeto.
<code>protected void finalize()</code>	Chamado pelo garbage collector quando não houver mais referências para o objeto (que pode nunca acontecer durante uma sessão da JVM)
<code>void notify()</code> <code>void notifyAll()</code>	Usado em métodos sincronizados para notificar threads que esperam acesso ao objeto
<code>void wait()</code> <code>void wait(long timeout)</code> <code>void wait(long timeout, int nanos)</code>	Usado em métodos sincronizados para suspender o thread atual até que outro thread chame <code>notify()</code> , ou o timeout expire.

## 5.2 Como estender Object

Classes que representam objetos que precisarem fazer cópias de si próprios devem implementar `clone()`. Objetos que participam de coleções, listas, mapas e conjuntos devem implementar `equals()` e `hashCode()`. Para que o objeto tenha uma representação em forma de String que reflete seu estado, deve-se redefinir `toString()`.

Esses são quatro métodos que qualquer classe Java poderá precisar redefinir. Dentre eles os mais importantes são `equals()` e `hashCode()` já que a sua ausência pode ser fonte de bugs em aplicações que usam tabelas de hash. É uma boa prática implementá-los em qualquer classe. Implementar `toString()` é importante para que o objeto possa ser convertido automaticamente numa representação String quando for concatenado.

Todos esses métodos devem ser declarados com `@Override`.

### 5.2.1 `toString()`

Este método por default imprime o nome da classe e o endereço de memória no heap onde o objeto está armazenado. É a melhor representação String para um Object, mas provavelmente não é para um Pedido, ou Endereço, ou Ponto. Um objeto Ponto provavelmente será melhor representado pela notação clássica de um ponto: (x,y):

```
@Override public String toString() {
    return "("+this.x+","+this.y+");"
```

### 5.2.2 `equals()`

Por default, um Object é igual a outro Object se os dois estiverem ocupando o mesmo endereço na memória. Ou seja, a implementação default compara os dois objetos com “==”. Não redefinir este método deixa `equals()` igual a ==.

Mas um ponto na posição (5,2) é “igual” a outra instância de Ponto com x=5 e y=2. O critério de igualdade é algo que depende do objeto e sua abstração. Um programa pode considerar que dois círculos de raio igual são iguais, enquanto outro poderá levar em conta também a posição. Um programa pode considerar que um objeto é igual a outro se ambos tiverem um mesmo ID (ex: chave primária), mesmo que os dados sejam diferentes. Um pode ser o objeto em versão anterior, e o outro conter os dados que irão atualizá-lo. Portanto, é necessário que cada objeto redefina seu `equals()` de acordo com os critérios definidos pela aplicação.

No caso da nossa abstração de Ponto, ele pode retornar true se os x e y dos dois pontos forem matematicamente equivalentes (tenham os mesmos valores de x e y):

```

@Override
public boolean equals(Object outro) {
    if(outro instanceof Ponto) {
        Ponto p = (Ponto) outro;
        return p.x == this.x && p.y == this.y;
    }
    return false;
}

```

### 5.2.3 hashCode()

Ao organizar uma coleção de objetos, um algoritmo de hash irá distribui-los em “balde” diferentes de acordo com um número informado por cada objeto. Para comparar uma instância com um dos objetos da coleção, primeiro o balde correspondente é recuperado, depois os objetos que pertencem a esse balde são comparados um a um com a instância.

Para que esse algoritmo simplesmente funcione, é preciso que dois objetos iguais retornem o mesmo código, ou `hashCode()`. Se isto nunca ocorrer, pode haver duplicação e um objeto armazenado poderá nunca mais ser recuperado em uma coleção.

Para que o algoritmo funcione eficientemente, é preciso que a distribuição dos códigos de *hash* seja tal que os baldes fiquem preenchidos mais ou menos uniformemente. Não é eficiente ter todos os objetos em um único balde. Em geral, `hashCodes` podem ser gerados e contém um número multiplicado por um *número primo*, para evitar que os números sejam múltiplos de 2 e caiam todos em um balde só. É comum multiplicar por 31, devido ao tamanho máximo do *int* ( $2^{32}$ ).

Para garantir que dois objetos iguais tenham o mesmo `hashCode`, os mesmos atributos usados para `equals()` devem ser usados no `hashCode()`.

Um bom `hashCode()` para *Ponto* (usando os mesmos dados de *equals* e número primo) pode ser:

```

@Override
public int hashCode() {
    return this.x * this.y * 31;
}

```

Não implementar `equals()` e `hashCode()` para objetos que serão armazenadas em coleções pode introduzir bugs difíceis de encontrar, principalmente se os objetos forem usados em frameworks complexos (ex: JPA, JSF) onde a causa do problema não seja facilmente percebida.

### 5.2.4 clone()

O método `clone()` deve ser implementado se o objeto precisar fazer cópias de si próprio. Uma vez implementado, é possível copiar um objeto usando:

```
Objeto copia = obj.clone();
```

Como `clone()` é *protected*, ele não pode ser usado por outras classes. É preciso que a classe que o herda o chame dentro de sua própria implementação. Simplesmente redefinir o método `clone()`, chamando-o na superclasse e trocando o modificador *protected* por *public* não funciona:

```

@Override
public Object clone() {
    return super.clone(); // não funciona - precisa ser Cloneable e capturar exceção!
}

```

Primeiro é preciso que a classe que implementa `clone()` declare implementar a interface `java.lang.Cloneable`:

```
public class Ponto implements Cloneable {}
```

Além disso, é preciso capturar a exceção `CloneNotSupportedException` dentro de um bloco *try*. Neste caso não é permitido usar *throws*, pois o método original não declara exceções (portanto o

método que sobrepõe não pode declará-las). Logo, a implementação mínima para implementar `clone()` é chamar a implementação original dentro desse bloco:

```
@Override
public Object clone() {
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        return null;
    }
}
```

Que é suficiente para duplicar um Ponto.

Mas se usarmos essa mesma estratégia com Linha não vai funcionar. A classe Linha possui *duas referências* para objetos Ponto. O `clone()` faz uma cópia rasa, e não faz a cópia dos objetos referenciados. O resultado será outro objeto Linha mas com os mesmos pontos compartilhados. Se um dele for alterado, as alterações irão se refletir no outro objeto Linha.

Para implementar `clone()` em linha, precisamos primeiro fazer a cópia rasa, e depois copiar cada referência. Como `clone()` já foi implementado em Ponto, é só chamar `clone()` em cada ponto:

```
public class Linha {
    ...
    @Override
    public Object clone() {
        try {
            Linha linha = (Linha)super.clone();
            linha.a = (Ponto)a.clone();
            linha.b = (Ponto)b.clone();
            return linha;
        } catch (CloneNotSupportedException e) {
            return null;
        }
    }
}
```

## 6 Classes parametrizadas

Várias classes em Java representam tipos que são associados a outros. Isto é comum em coleções de objetos, por exemplo. Para criar uma classe desse tipo é preciso passar um parâmetro de tipo entre `<...>` durante a declaração. Por exemplo, a classe abaixo representa um Pacote de alguma coisa. Representamos o alguma coisa pela letra E, de Elemento. E representa um tipo.

```
public class Pacote<E> {

    private E conteudo;

    public Pacote(E conteudo) {
        this.conteudo = conteudo;
    }
    public E getConteudo() {
        return conteudo;
    }
    public void setConteudo(E conteudo) {
        this.conteudo = conteudo;
    }
    public String toString() {
        return conteudo.toString();
    }
}
```

Com essa classe, podemos criar Pacotes de Strings, Pacotes de Integer, e assim por diante:

```
Pacote<String> pacote1 = new Pacote<>("Uma frase"); // Elemento é String
Pacote<Integer> pacote2 = new Pacote<>(12345); // Elemento é Integer
```

E recuperar objetos de tipos diferentes usando os mesmos métodos, sem precisar fazer cast:

```
int numero = pacote2.getConteudo();
String txt = pacote1.getConteudo();
```

Classes parametrizadas podem ter vários parâmetros. É incomum, porém, que tenham muitos. A classe Etiqueta abaixo associa um Tipo 1 que representa um código, a um Tipo 2 que representa um item. Código e Item podem ser implementados por qualquer classe:

```
public class Etiqueta<T1, T2> {
    private T1 codigo;
    private T2 item;

    public void setCodigo(T1 codigo) {
        this.codigo = codigo;
    }
    public void setItem(T2 item) {
        this.item = item;
    }
    public T1 getCodigo() {
        return codigo;
    }
    public T2 getItem() {
        return item;
    }
    public String toString() {
        return "codigo=" + codigo + ", item=" + item;
    }
}
```

Por exemplo, podemos criar um código que é um Integer, e um item que é um Pacote que contém um String:

```
Etiqueta<Integer, Pacote<String>> etiqueta1 = new Etiqueta<>();
etiqueta1.setCodigo(123);
etiqueta1.setItem(pacote1);
```

Ou ainda um código que é String e que contém um item que é um array de ints:

```
Etiqueta<String, int[]> etiqueta2 = new Etiqueta<>();
etiqueta2.setCodigo("456");
etiqueta2.setItem(new int[]{1,2,3});
```

Novamente, os métodos retornam os tipos correspondentes, sem precisar de cast:

```
int cod1 = etiqueta1.getCodigo();
String cod2 = etiqueta2.getCodigo();
Pacote<String> item1 = etiqueta1.getItem();
int[] item2 = etiqueta2.getItem();
```

A sintaxe de tipos parametrizados ou genéricos é bem abrangente, e permite restringir mais os tipos que podem ser usados em cada caso. O que vimos até agora é suficiente para usar a maior parte dos tipos parametrizados em Java.

Um exemplo de tipo parametrizado é Class<E>, que é o valor retornado pelo método getClass(), de Object. Se usarmos em Ponto, ele será:

```
Class<Ponto> clazz = p1.getClass();
```

## 6.1 Comparable<T>

O método `equals()` permite que um objeto seja comparado com outro em termos de igualdade, mas não oferece nenhuma informação para ordenação. Se objetos precisam ser ordenados, eles devem implementar a interface `Comparable<T>`. Esta interface possui um método:

```
int compareTo(T obj);
```

O método deve *retornar*:

- zero se os objetos forem iguais,
- um valor *menor que zero* que o objeto recebido for menor que o atual, e
- um valor *maior que zero* se o objeto recebido for maior.

Um objeto que implementa `Comparable<T>` pode ser passado para métodos que ordenam objetos, como o método `Arrays.sort()`, que é usado para ordenar arrays. Veja abaixo um exemplo de implementação em Ponto (para possibilitar o ordenamento de pontos pela sua distância até a origem):

```
public class Ponto implements Comparable<Ponto> {
    protected int x;
    protected int y;
    ...

    public double distancia() {
        return Math.sqrt(x * x + y * y);
    }

    @Override public int compareTo(Ponto p) {
        return (int)p.distancia() - (int)this.distancia();
    }
}
```

O cast antes de cada chamada ao método `distancia()` trunca as casas decimais, para que o método possa retornar `int`.

## 6.2 Comparator<T>

Se não for possível implementar Comparable, ainda é possível ordenar arrays de objetos usando um algoritmo de comparação externo. Para isto é preciso criar uma classe que implemente `java.util.Comparator<T>`. Esta interface possui um método:

```
public int compare(T obj1, T obj2);
```

As regras são as mesmas de `compare`. Zero se forem iguais. Valor menor que zero se `obj1 < obj2` e maior que zero se for o contrário. Um array de objetos `T` agora pode ser comparado em um método `sort()` de dois argumentos, passando um `Comparator<T>` como segundo argumento.

Podemos implementar um `Comparator` que ordene pontos pelo valor de X apenas. Como Ponto já tem um `Comparator`, este comparador terá que ser selecionado explicitamente quando usado:

```
import java.util.Comparator;

public class PontoXComparator implements Comparator<Ponto>{

    @Override public int compare(Ponto o1, Ponto o2) {
        return (int)o1.distancia() - (int)o2.distancia();
    }
}
```

Exploraremos o uso de comparadores na próxima seção (coleções, strings e arrays).

# 4 Arrays e strings

---

<b>1</b>	<b>Arrays</b>	<b>1</b>
<b>1.1</b>	<b>Criação</b>	<b>1</b>
<b>1.2</b>	<b>Acesso</b>	<b>2</b>
<b>1.3</b>	<b>Iteração</b>	<b>3</b>
<b>1.4</b>	<b>Arrays de arrays</b>	<b>3</b>
<b>1.5</b>	<b>Exceções</b>	<b>5</b>
<b>1.6</b>	<b>Cópia</b>	<b>5</b>
<b>1.7</b>	<b>Utilitário Arrays</b>	<b>5</b>
1.7.1	Cópia	6
1.7.2	Ordenação	6
1.7.3	Representação String	6
<b>2</b>	<b>Strings</b>	<b>7</b>
<b>2.1</b>	<b>Criação de strings</b>	<b>7</b>
<b>2.2</b>	<b>Operações com Strings</b>	<b>7</b>
2.2.1	Concatenação	8
2.2.2	Expressões regulares	8
<b>2.3</b>	<b>Modificação de Strings</b>	<b>9</b>
2.3.1	Métodos de StringBuilder	9

## 1 Arrays

Arrays, ou *vetores*, são objetos em Java, embora não sejam representados por uma classe. Um tipo declarado seguido de colchetes, é um tipo de array:

```
int[] arrayDeInt;
Ponto[] arrayDeObjetos;
String[] arrayDeString;
```

Arrays são imutáveis e têm tamanho fixo. São a maneira mais eficiente de organizar uma coleção de valores ou objetos em Java.

### 1.1 Criação

Para criar um array é necessário alocar memória no heap, usando o operador *new*. Um array tem *tamanho fixo* determinado pela quantidade de elementos. Essa quantidade é fixa e deve ser informada no momento em que o array é criado. Abaixo um exemplo de criação de um array de inteiros (declarado anteriormente):

```
arrayDeInt = new int[5];
```

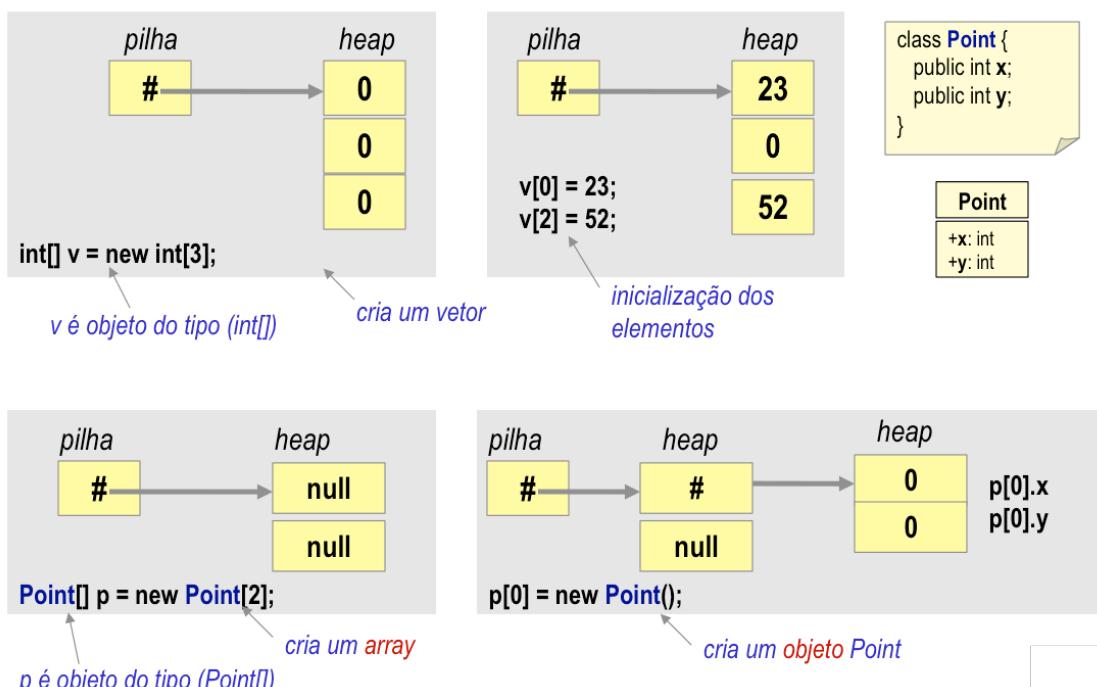
O array criado possui 5 inteiros, de 32 bits cada. Esse espaço é alocado no heap e o endereço é retornado e armazenado na referência arrayDeInt. O array foi criado e seus componentes foram inicializados com o valor 0.

Um array de objetos é similar:

```
arrayDeObjetos = new Ponto[3];
```

Foi alocado no heap espaço para armazenar três outros endereços de memória no heap, que serão os endereços dos objetos. Mas nenhum objeto foi armazenado. Os valores são inicializados com 0, mas como o tipo é um objeto, o valor traduz-se como null.

O desenho abaixo ilustra e compara as diferenças na alocação de memória para um array de objetos e um array de tipos primitivos:



Arrays também podem ser criados usando uma notação literal logo após serem declarados:

```
String[] arrayDeString = {"gato", "cachorro", "lagarto", "aranha"};
```

Isto aloca espaço no heap para quatro endereços (String é objeto), cria os quatro strings e armazena os endereços deles nas quatro posições, depois devolve o endereço para a referência arrayDeString, que poderá manipular o array.

Se a inicialização explícita ocorrer separadamente da declaração do tipo, é necessário usar new da forma:

```
arrayDeObjetos = new Ponto[] {new Ponto(5,5), new Ponto(3,2), new Ponto()};
```

Existem outras formas de criar arrays a partir de strings e coleções.

## 1.2 Acesso

Uma vez criado, os elementos de um *array* podem ser acessados pelo seu índice, lidos e alterados. O índice inicia em zero, portanto um *array* de 4 elementos tem índices 0, 1, 2 e 3:

```
System.out.println(arrayDeString[2]); // imprime lagarto
arrayDePontos[1].setX(50); // chama método setX(50) no segundo objeto do array
```

O valor de um elemento pode ser alterado com uma operação de atribuição:

```
arrayDeInt[0] = 23;
arrayDeInt[1] = 46;
arrayDeInt[9] = 230;
```

O array (de cinco inteiros) agora contém os valores:

```
{23, 46, 0, 0, 230}
```

Todo array possui uma variável pública chamada *length*, que informa o seu tamanho. A variável comprimento abaixo armazena o valor 4:

```
int comprimento = arrayDeInt.length;
```

### 1.3 Iteração

Para *iterar* por todos os elementos de um array pode-se usar um for, incrementando o índice, até que ele seja menor que *length*. O loop a seguir imprime todos os elementos de arrayDeString, numerados com base no índice:

```
for(int i = 0; i < arrayDeString.length; i++) {
    System.out.println((i+1) + ":" + arrayDeString[i]);
}
```

Este outro exemplo usa três arrays de tamanho igual para criar um quarto array de objetos, contendo dados dos três:

```
double[] latitudes = {-6.88, -9.01, -16.7, -23.5};
double [] longitudes = {-38.5, -42.7, -49.4, -46.8};
String[] nomes = {"Cajazeiras", "Sao Raimundo Nonato", "Goiânia", "São Paulo"};
assert latitudes.length == nomes.length && nomes.length == longitudes.length;

Localidade[] localidades = new Localidade[nomes.length];
for(int i = 0; i < nomes.length; i++) {
    localidades[i] = new Localidade();
    localidades[i].setLatitude(latitudes[i]);
    localidades[i].setLongitude(longitudes[i]);
    localidades[i].setNome(nomes[i]);
}
```

No final, o array de objetos *Localidade* está preenchido.

Se o índice não for necessário, e o objetivo for simplesmente iterar pelos elementos do array, pode-se usar o for-each. No exemplo abaixo, os elementos do array de localidades, preenchidos no exemplo acima, são impressos:

```
for(Localidade loc : localidades) {
    System.out.println(loc);
}
```

### 1.4 Arrays de arrays

Um array pode conter outro array. Por exemplo, dois arrays de ints:

```
int[] arrayDeInt1 = {1,2,3,4};
int[] arrayDeInt2 = {5,5};
```

Podem ser agrupados em um array. O tipo do array segue a mesma notação *Tipo[]*. Como o tipo é *int[]*, ele é declarado como *int[][]*:

```
int[][] arrayDeArrayDeInt = {arrayDeInt1, arrayDeInt2};
```

ou

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

ou ainda

```
int[][][] arrayDeArrayDeInt = new int[2][]; // inicializa duas posições, com null
```

Esta última declaração só cria o array externo. É preciso criar os dois arrays de int internos, com 4 e 2 elementos:

```
arrayDeArrayDeInt[0] = new int[4]; // inicializa pos 0 com array de 4 posições null
arrayDeArrayDeInt[1] = new int[2]; // inicializa pos 1 com array de 2 posições null
```

Em seguida, é necessário preencher os arrays internos:

```
arrayDeArrayDeInt[0][0] = 1;
arrayDeArrayDeInt[0][1] = 2;
arrayDeArrayDeInt[0][2] = 3;
arrayDeArrayDeInt[0][3] = 4;
arrayDeArrayDeInt[1][0] = 5;
arrayDeArrayDeInt[1][1] = 5;
```

Também é possível atribuir o array interno diretamente:

```
arrayDeArrayDeInt[1] = new int[] {5,5};
```

Arrays de arrays podem ter qualquer profundidade, e podem ser de qualquer tipo. No exemplo abaixo, a declaração de três vetores de Pontos: um tridimensional (espaço), um bidimensional (matriz) e um unidimensional (vetor):

```
Ponto[][][] espaço;
Ponto[][] matriz;
Ponto[] vetor;
```

A inicialização precisa sempre determinar o tamanho do array externo:

```
Object[][][] square = new Object[15][];
Object[][][] cube = new Object[10][][];
```

Mas se os arrays dentro do array externo tiverem tamanhos iguais, eles podem ser inicializados em uma única declaração:

```
Ponto[][] matriz = new Ponto[5][2]; // todos os 5 sub-arrays tem 2 elementos
Ponto[][][] espaço = new Ponto[3][4][2]; // 3 sub com 4 elem, 4 sub-sub com 2 elem
```

Para iterar em arrays multidimensionais, pode-se usar for aninhados. Podem ser iterações com for indexado. O for abaixo preenche o array (ja inicializado) com vários objetos Ponto:

```
for(int i = 0; i < matriz.length; i++) { // array externo
    for(int j = 0; j < matriz[i].length; j++) { // cada array interno
        matriz[i][j] = new Ponto(i*10, j*10);
    }
}
```

Se o array não estiver totalmente inicializado, ocorrerá NullPointerException caso haja tentativa de acessar um índice não inicializado. Por exemplo, se for inicializado apenas o array externo:

```
Ponto[][] matriz = new Ponto[5][]; // array interno não inicializado!
```

Cada array interno precisará ser instanciado dentro do loop, antes de ser usado:

```
for(int i = 0; i < matriz.length; i++) { // array externo
    matriz[i] = new Ponto[2]; // instanciação de array interno
    for(int j = 0; j < matriz[i].length; j++) { // cada array interno
        matriz[i][j] = new Ponto(i*10, j*10);
    }
}
```

Para iteração sem a necessidade do índice, o for-each geralmente é mais compacto e mais legível:

```
for(Ponto[] pontos: matriz) // para cada array pontos, do array matriz
    for(Ponto ponto: pontos) // para cada ponto, do array de pontos
        System.out.println(ponto);
```

(Os blocos for, while e if *podem* ser usados sem as chaves, mas em geral isto não é considerado uma boa prática; exceções são expressões muito simples como esta).

## 1.5 Exceções

A tentativa de acessar um elemento de array inexistente causa uma exceção (erro de tempo de execução) do tipo *ArrayIndexOutOfBoundsException*:

```
int[] numeros = {1,2,3,4};
System.out.println(numeros[4]); // nao existe elemento 4!
```

A tentativa de acessar um array ainda não instanciado causa *NullPointerException*:

```
double[] precos;
precos[0] = 1.00; // o array ainda não existe!
```

## 1.6 Cópia

Arrays não são modificáveis. Isto não se refere ao seu conteúdo, que pode ser alterado, mas à sua estrutura (tipo) e tamanho. Como não é possível modificar o tamanho de um array, pode ser necessário extraír um conjunto dos seus elementos para criar um array menor, ou copiar o array em um array maior, que possa receber mais elementos.

A maneira mais eficiente (embora não seja a mais simples) de fazer uma cópia é usando *System.arraycopy()*:

```
static void arraycopy(origem_da_copia, offset,
                      destino_da_copia, offset,
                      numero_de_elementos_a_copiar)
```

Por exemplo, o trecho abaixo usa *arrayCopy* para encaixar o arrayUm abaixo no meio do arrayDois :

```
int[] arrayUm = {12, 22, 3};
int[] arrayDois = {9, 8, 7, 6, 5};
System.arraycopy(arrayUm, 0, arrayDois, 1, 2);
```

O resultado será o array dois contendo:

```
{9, 12, 22, 6, 5}
```

Tipicamente as cópias são mais simples, copiando arrays menores para arrays vazios, para permitir a adição de elementos a mais, ou copiando os elementos não-nulos de um array para um array menor, para utilizar o espaço mais eficientemente. Para copiar um array menor inteiro para dentro de outro maior:

```
System.arraycopy(arrayMenor, 0, arrayMaior, 0, arrayMenor.length);
```

## 1.7 Utilitário Arrays

A classe Arrays, do pacote *java.util* contém vários métodos estáticos para realizar operações comuns com arrays:

- **binarySearch(array, valor)** – pesquisa um valor no array e retorna sua primeira posição
- **copyOf(array, comprimento)** – retorna uma cópia do array truncado no comprimento.
- **copyOfRange(array, inicio, fim)** – retorna uma cópia de uma faixa do array, iniciando no índice inicio, até antes do índice fim.

- **equals()** – retorna true se os arrays forem iguais (se eles tiverem o mesmo tamanho e contiverem os mesmos elementos na mesma ordem) – deve ser usado para arrays de tipos primitivos.
- **deepEquals(Object[], Object[])** – retorna true se os arrays de profundidade arbitrária forem idênticos – deve ser usado para arrays de objetos.
- **fill(array, valor)** – preenche o array com um valor
- **hashCode(array)** – retorna um hashCode para o array (útil para implementar hashCode() em objetos)
- **sort(array)** – ordena array de tipos primitivos ou de objetos que implementam Comparable<T>
- **sort(array, Comparator<T>)** – ordena array de acordo com algoritmo passado como segundo argumento
- **toString(array)** – imprime uma representação do array como string.

### 1.7.1 Cópia

`Arrays.copyOf()` e `Arrays.copyOfRange()` são uma alternativa a `System.arraycopy()` para extrair porções de um array:

```
int[] numeros = {2, 6, 7, 8, 9, 23, 34, 122};
int[] slice = Arrays.copyOf(numeros, 3);
System.out.println(Arrays.toString(slice)); // [2, 6, 7]

int[] range = Arrays.copyOfRange(numeros, 2, 5);
System.out.println(Arrays.toString(range)); // [7, 8, 9]
```

### 1.7.2 Ordenação

Arrays podem ser ordenados com `Arrays.sort()`. Isto é automático para tipos primitivos:

```
int[] numeros = {7,23,9,122,6,34,8,2};
Arrays.sort(numeros);
System.out.println(Arrays.toString(numeros)); // [2, 6, 7, 8, 9, 23, 34, 122]
```

Objetos que precisam ser ordenados, devem implementar a interface `Comparable<T>` e o método `compareTo()` para informar os critérios de ordenação que serão usados.

```
Ponto[] pontos = {new Ponto(3,4), new Ponto(1,1), new Ponto(5,1)};
Arrays.sort(pontos); // usa compareTo() de Ponto
```

Classes da API Java que são ordenáveis implementam `Comparable`. String ordena de forma case-sensitive por default, sem levar em conta acentos (ou seja, A e Z vêm antes de a e z, que vêm antes de qualquer caractere acentuado):

```
String[] textos = {"sapo", "Zebra", "anta", "Águia", "Antílope", "bode"};
Arrays.sort(textos);
System.out.println(Arrays.toString(textos)); // [Antílope, Zebra, anta, bode, sapo, Águia]
```

Para substituir o algoritmo nativo de ordenação, ou para ordenar objetos que não implementam `Comparable`, é possível criar um `java.util.Comparator` contendo o algoritmo a ser usado e passar como segundo argumento de `Arrays.sort()`:

```
Comparator<String> algoritmo = new IgnoreCaseCollationComparator();
Arrays.sort(textos, algoritmo); // [Águia, anta, Antílope, bode, sapo, Zebra]
```

### 1.7.3 Representação String

Uma representação String do array é obtida através de `Arrays.toString()`:

```
System.out.println(Arrays.toString(latitudes));
```

Imprime:

```
[-6.88, -9.01, -16.7, -23.5]
```

## 2 Strings

Strings são objetos que representam cadeias de caracteres. Em Java, strings são imutáveis e têm um tratamento diferente dos demais objetos. Podem ser criados sem usar new, através da atribuição de um literal, e são mantidos em um pool para reutilização, já que são imutáveis.

Para alterar strings, Java inclui a classe *StringBuilder* (a classe *StringBuffer* também pode ser usada para este fim, mas ela é menos eficiente porque seus métodos são sincronizados).

### 2.1 Criação de strings

Strings podem ser criados da mesma forma como se cria objetos comuns:

```
String s = new String("I am a String!");
```

ou da forma mais usual:

```
String s = "I am a String!";
```

Ambos produzem o mesmo resultado só que o último tem um tratamento especial e é tratado mais como um tipo básico que como um String. Se você comparar usando “==” dois Strings criados usando *new*, eles serão apontados como diferentes (a referência foi comparada), mas se os dois tiverem sido criados usando literais, o sistema irá reutilizar a mesma referência, e eles geralmente serão considerados iguais.

Mas Strings devem sempre ser comparados com *equals()*, nunca com *==*. A comparação com *==* é uma fonte comum de bugs.

### 2.2 Operações com Strings

Alguns métodos da classe String estão listados abaixo. Nenhum método altera o string no qual ele é chamado, mas retorna um string novo:

- char **charAt(int index)**: retorna o caractere no local indicado por index (a primeira posição é 0).
- String **concat(String str)**: concatena o String atual com outro, retornando um novo String (mesmo que usar “+”).
- boolean **endsWith(String sufixo)**: retorna true se o String termina com o sufixo passado como parâmetro.
- boolean **startsWith(String prefixo)**: retorna true se o String começa com o prefixo passado como parâmetro.
- int **indexOf(String str, int inicio)**: retorna o índice do início do String str, a partir da posição inicio. Este método também pode receber como primeiro argumento um char e não ter o segundo argumento, na versão com char ou na versão com String.
- int **lastIndexOf(String str, int fim)**: faz o mesmo que *indexOf* só que de trás para frente. Também tem as mesmas variações sobrecarregadas que o método anterior. O fim não faz parte do String, que vai de inicio a fim-1.
- int **length()**: retorna o comprimento do String em caracteres.
- public String **replace(char antigo, char novo)**: troca o caractere antigo por um novo e devolve um novo String.
- String **substring(int início, int fim)**: retorna um novo String iniciando na posição início do String atual e terminando em fim-1.
- String **toLowerCase()** e public String **toUpperCase()** retornam um novo String em caixa-alta ou caixa-baixa respectivamente.

- int **compareToIgnoreCase**(String str) – compara dois Strings ignorando formato maiúsculo/minúsculo e retorna valor maior que zero se o string tiver ordem maior que a atual, menor que zero se vier antes, e igual a zero se a ordem for igual.
- boolean **contains**(String str) – retorna true se o string contém o substring passado como argumento.
- String **format**(String template, Object[] args) – Recebe um template no estilo sprintf do C para preenchimento pelo array de objetos passado no segundo argumento.
- boolean **regionMatches**(int offset, String outro, int outroOffset, int len)
- String **replaceAll**(String regex, String substituição) – retorna um string com substituições determinadas por uma expressão regular.
- char[] **toCharArray**() – retorna um array de chars com os caracteres do string.
- String[] **split**(String regex) – retorna um array formado pela divisão do string de acordo com uma expressão regular.
- String **trim()** – retorna um string sem espaços brancos antes e depois.

## 2.2.1 Concatenação

A concatenação de strings sempre cria novos objetos. O uso de “+” é equivalente ao uso de `concat()`. No código abaixo são criados seis objetos para realizar a concatenação:

```
String a = "um", b = "dois", c = "tres";
String resultado = a + b + c + "quatro";
```

Portanto, um código que realiza muitas concatenações dessa forma poderá causar problemas de performance devido a grande quantidade de objetos construídos e destruídos, que irão forçar a execução antecipada do *Garbage Collector*. Concatenações em String podem ser substituídas por operações com `StringBuilder`, que são mais eficientes.

## 2.2.2 Expressões regulares

Em Java existe uma API de expressões regulares no pacote `java.util`, que permite transformações e pesquisas sofisticadas em Strings. Alguns métodos de `String` recebem expressões regulares. Um dos mais usados é o método `split()`, que gera um array novo resultante da aplicação da expressão regular.

A expressão pode consistir de um único caractere delimitador, por exemplo:

```
String itens = "um, dois, tres, quatro";
String[] itensArray = itens.split(",");
String item2 = itensArray[1].trim(); // dois
```

Mas é importante lembrar que é um Regexp e não esquecer de usar escapes para símbolos especiais, como por exemplo o ponto “.”, que representa qualquer caractere em Regexp. Se ele deve representar realmente um ponto, precisa ser precedido de um escape, que em Regexp é uma contrabarra. Isto representa um ponto em Regexp:

\.

O problema é que a contrabarra também é escape para strings. Então para poder fazer um `split()` usando o delimitador “.” em Java, é preciso usar *duas* contrabarras:

```
String nomeArquivo = "imagem.jpg";
String[] nomeExt = nomeArquivo.split("\\.");
String nome = nomeExt[0]; // imagem
String extensão = nomeExt[1]; // jpg
```

## 2.3 Modificação de Strings

A classe `StringBuilder` possui vários métodos que *modificam* strings na memória. Mas `StringBuilders` não podem ser usados. É preciso transformá-los em `Strings` imutáveis antes do uso:

```
StringBuilder builder = new StringBuilder("texto");
builder.append(" alterado");
String imutavel = builder.toString(); // transforma em string
```

É muito mais eficiente fazer transformações e concatenações com `StringBuilder`, para depois converter em `String`. Isto evita a alocação desnecessária de memória na criação de muitos objetos.

Além de receber um string inicial, o `StringBuilder` pode também ser construído passando um tamanho inicial ou vazio:

```
StringBuilder b1 = new StringBuilder();
StringBuilder b2 = new StringBuilder(1000);
```

Os métodos de `StringBuilder` retornam a própria instância, permitindo que sejam concatenados:

```
builder.append(" com").append(" quatro").append(" comandos");
```

### 2.3.1 Métodos de `StringBuilder`

`StringBuilder` contém vários dos métodos que existem em `String`, com a diferença que não apenas retornam um `StringBuilder` alterado, mas também alteram o buffer na memória. Os métodos que retornam `StringBuilder` podem ser concatenados. Abaixo estão alguns dos métodos de `StringBuilder`:

- `StringBuilder insert(int posição, String str)`,
- `StringBuilder delete(int inicio, int fim)`
- `StringBuilder append(String str)`
- `void setCharAt(int posição, char caracter)`
- `StringBuilder reverse()`
- `StringBuilder replace(int start, int end, String str)`
- `void setLength(int novoTamanho)`.

# 5 Coleções

<b>1</b>	<b>Coleções</b>	<b>1</b>
<b>2</b>	<b>Iterators</b>	<b>3</b>
<b>3</b>	<b>Interfaces</b>	<b>3</b>
<b>3.1</b>	<b>Collection&lt;E&gt;</b>	<b>3</b>
<b>3.2</b>	<b>List&lt;E&gt;</b>	<b>4</b>
<b>3.3</b>	<b>Set&lt;E&gt;</b>	<b>4</b>
<b>3.4</b>	<b>Map&lt;K,V&gt;</b>	<b>5</b>
3.4.1	Map.Entry<K,V>	5
3.4.2	Método hashCode()	6
<b>3.5</b>	<b>Queue&lt;E&gt; e Deque&lt;E&gt;</b>	<b>6</b>
<b>4</b>	<b>Classes utilitárias</b>	<b>7</b>

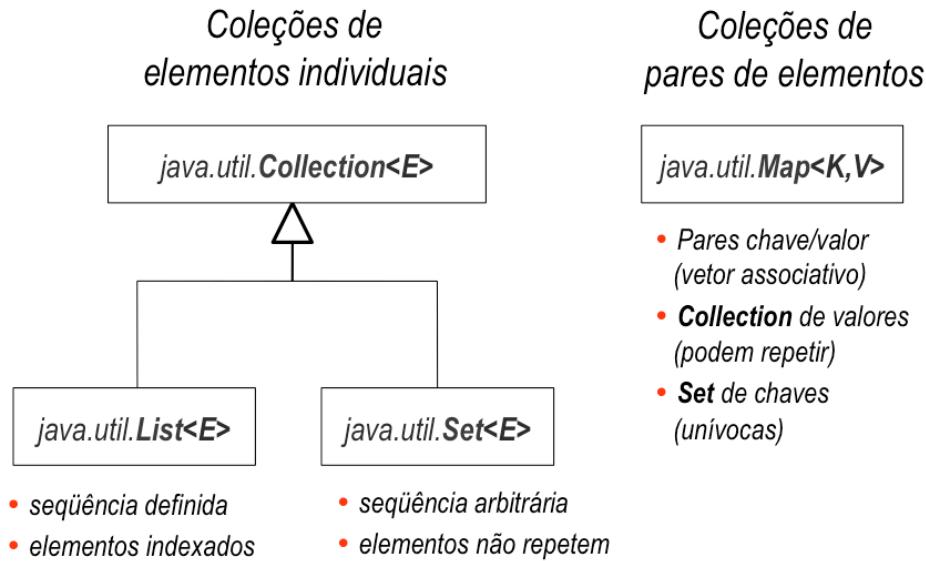
Coleções são estruturas de dados de propósito geral: vetores, conjuntos, pilhas, arvores binárias, mapas de hash, filas, etc. que servem para agrupar objetos. Java já permite a organização de coleções de objetos em arrays, mas eles são pouco flexíveis e requerem a criação de operações de cópia para redimensioná-los. Coleções oferecem estruturas redimensionáveis, modificáveis que encapsulam essas operações e podem ser usadas com uma interface mínima.

Embora seja possível construir aplicações Java sem usar esta API, ela é essencial e usada na grande maioria das APIs e frameworks do Java SE e Java EE. É uma API fundamental e consiste principalmente de classes e interfaces parametrizadas (genéricas) que aceitam qualquer tipo de objeto.

Este capítulo explora as classes e interfaces essenciais da API de coleções (que são usados na maior parte das aplicações).

## 1 Coleções

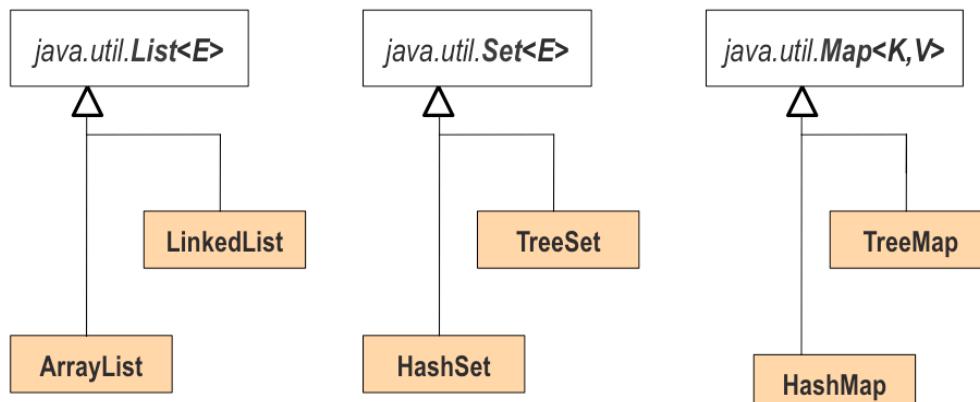
Existem várias interfaces e implementações de coleções no pacote *java.util*. As mais importantes derivam das interfaces *Collection* e *Map* que estão ilustradas abaixo:



Esta é uma hierarquia de interfaces. Podemos usá-la para declarar diferentes tipos de coleções:

```
List<Integer> numeros;
Map<String, Ponto> mapaDePontos;
```

Para criar uma coleção, é preciso escolher uma implementação. Existem várias. Devem ser escolhidas de acordo com a forma como serão usadas. Algumas são mais eficientes para dados ordenados, outras melhores para dados que serão lidos mas raramente alterados, e assim por diante. Estas são algumas (talvez as principais) implementações de `List`, `Set` e `Map`:



Algoritmos de hash são eficientes para leitura e gravação e geralmente são a escolha mais comum para implementar um `Map`:

```
Map<String, Ponto> mapaDePontos = new HashMap<>();
```

Conjuntos (`Sets`) não são indexados como `Lists`, e para recuperar dados que podem ser ordenados, poderá mais eficiente implementar um `TreeSet` se alterações não forem muito frequentes (outra opção é ser mais específico e usar a interface `SortedSet`):

```
Set<Circulo> circulos = new TreeSet<>();
```

`Lists` são como arrays, que podem ser modificados, adicionar e remover elementos durante o uso. Talvez os elementos sejam encadeados e um `LinkedList` seja melhor (ou mesmo uma outra interface, como `Queue`, que também é implementada por `LinkedList`), mas em listas de propósito geral a implementação mais comum é `ArrayList`:

```
List<Integer> numeros = new ArrayList<>();
```

## 2 Iterators

Iterators são objetos que são usados para acessar todos os elementos de uma coleção. Eles mantém um controle sobre os objetos já visitados, e permitem que sejam extraídos um a um, em ordem arbitrária. Iterators podem ser usados para extrair elementos de qualquer coleção.

Java possui várias diferentes interfaces de *iterators*.

No pacote *java.sql* existe o *ResultSet*, que é um iterator para navegar em registros de um banco de dados.

No pacote *java.util* existem duas interfaces básicas: *Enumeration* e *Iterator*. Um iterator é fornecido através de um método de fábrica por uma coleção. A maior parte das coleções usam *Iterator*, mas algumas APIs antigas que usam coleções das primeiras versões de Java como *Vector* e *Hashtable* ainda usam *Enumeration*. O funcionamento é igual. A única diferença é a interface.

Interface Enumeration:

```
package java.util;
public interface Enumeration<E> {
    boolean hasMoreElements();
    E nextElement();
}
```

Interface Iterator:

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Alguns frameworks e APIs do Java EE (ex: Servlets) usam *Enumeration* em vez de *Iterator*.

Coleções que retornam um iterator implementam a interface *Iterable<T>* e seu método *iterator()*, que retorna um *Iterator<T>*. Qualquer coleção que implementa *Iterable* pode ser usada no for-each:

```
for(Object objeto : coleção) {
    // fazer algo com cada objeto
}
```

que é equivalente a este for tradicional:

```
for(Iterator it = coleção.iterator(); it.hasNext();) {
    // fazer algo com cada objeto
}
```

Ou seja, o for-each chama automaticamente o método *iterator()* de qualquer coleção que implementa *Iterable*.

## 3 Interfaces

### 3.1 Collection<E>

*Collection<E>* representa uma coleção arbitrária, não ordenada e que permite elementos duplicados. Abaixo a interface *Collection* destacando seus principais métodos, que têm nomes auto-explicativos:

```

public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[]);
    boolean add(E);
    boolean remove(Object);
    boolean containsAll(Collection<?>);
    boolean addAll(Collection<? extends E>);
    boolean removeAll(Collection<?>);
    boolean retainAll(Collection<?>);
    void clear();
    ...
}

```

Uma coleção é usada através de uma implementação concreta. Geralmente usa-se uma interface mais específica como *List* ou *Set*.

## 3.2 List<E>

*List* é um array modificável de propósito geral. Herda os métodos de *Collection* e adiciona alguns relacionados a coleções indexadas:

- void **add(int index, E o)**: adiciona objeto na posição indicada (empurra elementos existentes para a frente)
- E **get(int index)**: recupera objeto pelo índice
- int **indexOf(Object o)**: procura objeto e retorna índice da primeira ocorrência
- E **set(int index, E o)**: grava objeto na posição indicada (apaga qualquer outro que ocupava a posição).
- E **remove(int index)**
- ListIterator<E> **listIterator()**: retorna um iterator capaz de navegar com índices.

Exemplo de uso:

```

List<Ponto> lista = new ArrayList<>();
lista.add(new Ponto(12,34));
lista.add(new Ponto(66,77));
lista.add(new Ponto(90,32));
Ponto c3 = lista.get(2); // terceiro elemento

ListIterator<Ponto> it = lista.listIterator();
Ponto c = it.last(); // ultimo
Ponto a = it.previous(); // volta um

Ponto[] pontos = list.toArray(new Ponto[lista.size()]);

```

## 3.3 Set<E>

*Set* representa um conjunto matemático. Não possui valores repetidos. Acrescenta e altera alguns métodos de *Collection* para lidar com dados que não se repetem.

- boolean **add(Object)**: só adiciona o objeto se ele já não estiver presente (usa equals() para saber se o objeto é o mesmo)
- **contains(), retainAll(), removeAll(), ...**: redefinidos para lidar com restrições de não-duplicação de objetos (esses métodos funcionam como operações sobre conjuntos)

Sets dependem da correta implementação do método equals() de cada objeto que estiver armazenado no conjunto, caso contrario poderão ocorrer duplicações. Se a implementação usada for HashSet, é preciso que o hashCode() também esteja corretamente implementado.

Exemplo de uso:

```
Set<String> conjunto = new HashSet<>();
conjunto.add("Um");
conjunto.add("Dois");
conjunto.add("Tres");
conjunto.add("Um");
conjunto.add("Um");
Iterator<String> it = conjunto.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

Resultado da execução:

```
Um
Dois
Tres
```

## 3.4 Map<K,V>

Objetos Map são semelhantes a vetores mas, em vez de índices numéricos, usam chaves, que são objetos. As chaves (Keys) são unívocas (estão em um Set) e os valores (Values) podem ser duplicados (estão em uma Collection). As principais subclasses são HashMap e TreeMap.

Métodos

- void **put(K key, V value)**: acrescenta um objeto
- V **get (K key)**: recupera um objeto
- Set<K> **keySet()**: retorna um Set de chaves
- Collection<V> **values()**: retorna um Collection de valores
- Set<K> **entrySet()**: retorna um set de pares chave-valor contendo objetos representados pela classe interna *Map.Entry*

Como as chaves de um Map são um Set, Maps também dependem da correta implementação do método equals() de cada objeto que estiver armazenado no conjunto, caso contrario poderão ocorrer duplicações. Se a implementação usada for HashSet, é preciso que o hashCode() também esteja corretamente implementado.

Exemplos:

```
Map<String, Ponto> map = new HashMap<>();
map.put("um", new Ponto(3,4));
map.put("dois", new Ponto(9,9));
//
Set<String> chaves = map.keySet();
Collection<Ponto> valores = map.values();
//
Ponto c = map.get("dois");
```

### 3.4.1 Map.Entry<K,V>

Classe interna usada para manter pares chave-valor em qualquer implementação de Map. Seus principais métodos são:

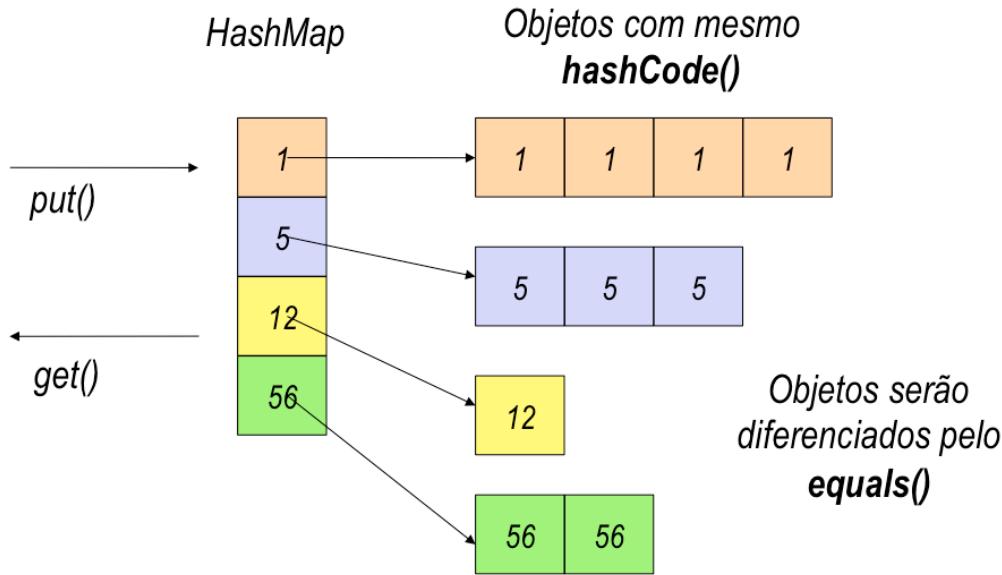
- K **getKey()**, retorna a chave do par;
- V **getValue()**, retorna o valor.

Exemplos:

```
Set<String> pares = map.entrySet();
Iterator<Ponto> entries = pares.iterator();
Map.Entry<String, Ponto> one = entries.next();
String chaveOne = one.getKey();
Ponto valueOne = one.getValue();
```

### 3.4.2 Método hashCode()

É fundamental que o hashCode() dos objetos que participam de uma implementação de hash (ex: HashSet, HashMap) implementem corretamente o hashCode() e equals(). Veja mais detalhes no capítulo sobre classes e objetos.



Existem várias implementações de Map, principalmente HashMap, por exemplo, para uso em ambientes concorrentes (ConcurrentHashMap), ou para situações em que são usadas referências temporárias (WeakHashMap).

## 3.5 Queue<E> e Deque<E>

Queue representa uma fila. Deque é uma fila de dois lados. São as principais interfaces para implementação de filas. Foi incluída na Collections API a partir do Java 5.0. A interface de Queue está listada abaixo:

```
interface Queue<E> extends Collection<E> {
    E element();           // lê mas não remove cabeça
    E peek();              // lê mas não remove cabeça
    E poll();               // lê e remove cabeça
    boolean offer(E o);   // insere elemento da fila
    E remove();             // remove elemento da fila
}
```

A ordenação depende da implementação utilizada.

Exemplo de uso:

```
Queue<String> fila = new LinkedList<String>();
fila.offer("D"); // cabeça, em fila FIFO
fila.offer("W");
fila.offer("X"); // cauda, em fila FIFO
System.out.println("Decapitando: " + fila.poll()); // D
System.out.println("Nova cabeça: " + fila.peek()); // W
```

A principal implementação usada é `LinkedList`, que é uma lista encadeada de propósito geral usada para **pilhas, filas e dequeus**.

## 4 Classes utilitárias

Além da classe `java.util.Arrays`, que possui vários métodos estáticos para trabalhar com arrays, existem as classes `java.util.Collections` e `java.util.Objects` que fornecem métodos similares para respectivamente objetos e coleções. Consulte a documentação sobre essas duas classes para mais detalhes.

# 6 Exceções

---

<b>1 Tipos de exceções</b>	<b>1</b>
<b>2 Como usar exceções</b>	<b>2</b>
<b>2.1 Como lançar</b>	<b>2</b>
<b>2.2 Como capturar</b>	<b>3</b>
2.2.1 Blocos try e catch	3
2.2.2 Bloco finally	3
2.2.3 Try com resources	4
<b>3 Como declarar</b>	<b>5</b>
<b>4 Como criar exceções</b>	<b>6</b>
<b>5 Reuso de exceções</b>	<b>7</b>
<b>5.1 Algumas exceções que devem ser reusadas</b>	<b>7</b>

Exceções são objetos usados para lidar com situações excepcionais. Quando um erro ou situação excepcional ocorre, uma exceção é instanciada e lançada. Ela causa o retorno de toda a cadeia de métodos que chamaram a exceção. Este processo só é interrompido se algum bloco *catch* pelo caminho conseguir capturar a exceção.

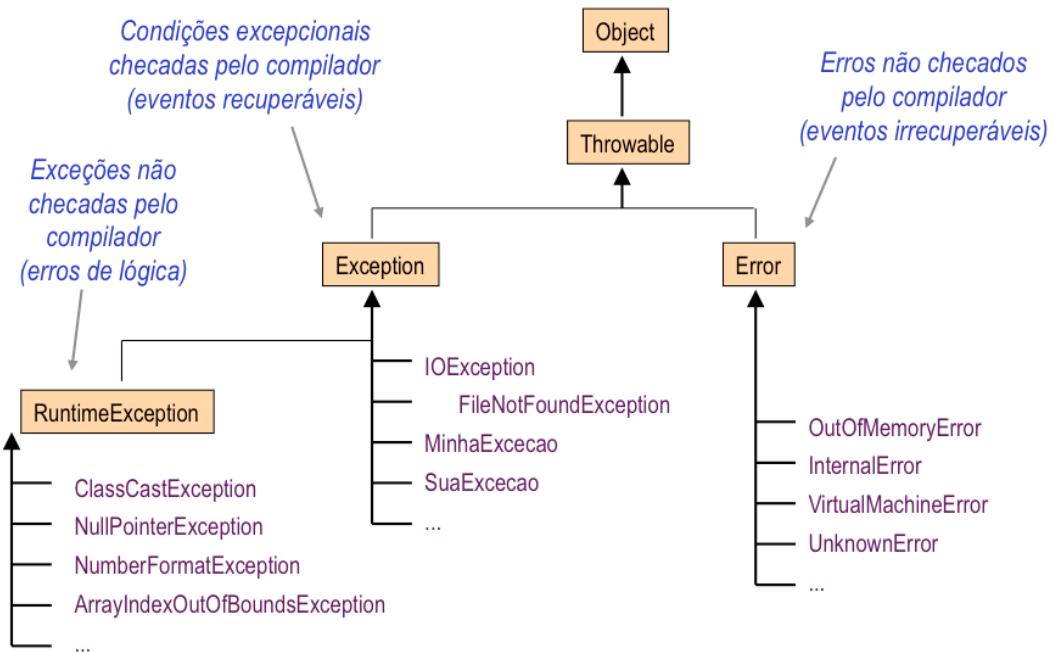
## 1 Tipos de exceções

Existem vários tipos de exceções, e elas podem ser lançadas em situações de erro causadas por bugs no programa, por falhas na infraestrutura do ambiente de execução e por problemas de comunicação entre a aplicação e algum serviço externo. Algumas exceções servem apenas para informar erros de tempo de execução e encerrar a aplicação, já que são erros irrecuperáveis. Outras são lançadas para que o sistema tente a recuperação de alguma forma.

De acordo com a especificação Java, existem três tipos de erros de tempo de execução que causam exceções:

1. *Erros de lógica de programação*. Ex: limites do vetor ultrapassados, divisão por zero. Estes erros devem ser corrigidos pelo programador
2. *Erros devido a condições do ambiente de execução*. Ex: arquivo não encontrado, rede fora do ar, etc. Fogem do controle do programador mas podem ser contornados em tempo de execução.
3. *Erros graves onde não adianta tentar recuperação*. Ex: falta de memória, erro interno do JVM. Fogem do controle do programador e não podem ser contornados

Essas diferentes situações são refletidas na hierarquia de classes de exceções, que tem como raiz a classe *java.lang.Throwable*.



*Throwable* possui duas subclasses: *Error* e *Exception*. *Error* representa erros geralmente irrecuperáveis, como falta de memória, estouro de pilha, falhas na JVM, etc. Ou seja, os erros do tipo 3.

*Exception* se divide em dois grupos. Um consiste dos descendentes de *RuntimeException*, que foram, na sua maioria, criados para identificar erros de lógica de programação. São as exceções do tipo 1.

O restante das exceções são chamadas de exceções checadas (*checked exceptions*) e representam condições excepcionais que podem acontecer em qualquer programa, e que não tem a ver com bugs nem falhas no sistema. Essas exceções representam o tipo 2, e precisam ser tratadas (capturadas ou declaradas).

## 2 Como usar exceções

### 2.1 Como lançar

Exceções são objetos. Para lançar uma exceção é preciso primeiro criar uma instância dela. Em seguida, a instância é passada como parâmetro da instrução *throw*. A informação mais importante que uma exceção preserva é o nome de sua classe, que geralmente já informa o que ocorreu. Outros parâmetros podem ser passados ao objeto antes do lançamento, mas essas informações costumam ser passadas no construtor. Portanto é comum criar e lançar a exceção usando uma única instrução:

```
throw new FileNotFoundException("Arquivo de configuração não foi encontrado");
```

Quando um *throw* acontece, o fluxo de execução é interrompido. O que acontece em seguida depende se o *throw* aconteceu em um bloco *try*, ou não. Se não, o método retorna imediatamente o controle ao ponto seguinte àquele de onde foi chamado. Se está em um bloco *try*, as cláusulas *catch* serão testadas em sequência até que alguma delas declare capturar uma exceção de tipo compatível com a que foi lançada. Se nenhum bloco *catch* capturar a exceção, um bloco *finally* se houver será executado e o método retorna. Se um bloco *catch* capturar a exceção, o método irá continuar sua execução depois do *try-catch*.

## 2.2 Como capturar

Código que potencialmente causa exceções deve estar em um bloco *try-catch*. O bloco *try* “tenta” executar, e se algo sair errado, uma ou mais cláusulas *catch* tentam pegar o problema e consertá-lo. Um bloco *finally* pode ser usado com ou sem *catch*, para executar código que precise ser executado sempre.

### 2.2.1 Blocos try e catch

A sintaxe do *try-catch* é a seguinte:

```
try {
    //... instruções ...
} catch (TipoExcecao1 ex) {
    //... faz alguma coisa ...
} catch (TipoExcecao2 ex) {
    //... faz alguma coisa ...
} catch (Exception ex) {
    //... faz alguma coisa ...
}
// ... continuação se a exceção for capturada
```

Os blocos *catch* são executados em ordem. Se alguma exceção for superclasse de outra, a superclasse precisa aparecer depois, ou capturará também as exceções da subclasse. É possível também (embora não seja uma boa prática) usar *Exception* como superclasse que captura todas as exceções. Neste caso apenas um *catch* é necessário:

```
try {
    //... instruções ...
} catch (Exception ex) {
    //... faz alguma coisa ...
}
// ... continuação se a exceção for capturada
```

Se todas as exceções que ocorrerem devem ser tratadas da mesma forma, pode-se usar um *multicatch*, que é uma alternativa mais recomendada que o *catch* com *Exception*:

```
try {
    //... instruções ...
} catch (TipoExcecao1 | TipoExcecao2 | Exception ex) {
    //... faz alguma coisa ...
}
// ... continuação se a exceção for capturada
```

Exceções *chequadas* precisam ser lançadas ou declaradas. A decisão em fazer uma coisa ou outra depende da responsabilidade do método onde ela ocorreu.

O bloco *try* pode ser abrangente e conter várias linhas de código e potencialmente causar várias exceções diversas em instruções diferentes, ou pode ser pequeno e pontual, cuidando de uma situação excepcional causada por um método ou instrução. A decisão de usar uma abordagem ou outra depende do nível de controle que se deseja ter sobre as exceções e recuperação da situação excepcional. Mesmo capturada, a exceção deixa o bloco *try* no ponto onde ocorreu. Se ela ocorreu no início do bloco, as instruções que vieram depois não serão mais executadas.

### 2.2.2 Bloco finally

O bloco *try* não pode aparecer sozinho. Deve ser seguido por pelo menos um *catch* ou por um *finally* que deve conter instruções que precisam se executadas independentemente da ocorrência ou não de exceções

```
try {
    // instruções: executa até linha onde ocorrer exceção
```

```

} catch (TipoExcecao1 ex) {
    // executa somente se ocorrer TipoExcecao1
} catch (TipoExcecao2 ex) {
    // executa somente se ocorrer TipoExcecao2
} finally {
    // executa sempre ...
}
// executa se exceção for capturada ou se não ocorrer

```

Situações típicas são o fechamento de recursos escassos que não são liberados automaticamente, como conexões de bancos de dados, streams, etc. Normalmente esses recursos são abertos no *try*, eventuais problemas são tratados em um ou mais blocos *catch*, e o *finally* fecha independente de falha ou sucesso.

No exemplo abaixo um método em algum momento tenta ler um arquivo.

```

private static void imprimir() throws IOException {
    System.out.println("1) Várias outras operações de impressão...");
    Reader input = null;
    try {
        System.out.println("2) Tentativa de ler o arquivo");
        input = new FileReader("texto.txt");

        byte[] buffer = new byte[1024];
        int len = input.read(buffer);
        while(len != -1){
            System.out.write(buffer, 0, len);
            len = input.read(buffer);
        }
        System.out.flush();
        System.out.println("3) Arquivo lido com sucesso");
    } catch (IOException e) {
        System.out.println("4) Erro de IO - arquivo não foi lido!");
    } finally {
        if(input != null){
            input.close();
            System.out.println("5) Stream foi fechado!");
        }
    }
    System.out.println("6) Continuação ... mais operações ...");
}

```

O código do bloco *try* será executado. Os cenários possíveis são:

- **Sucesso.** Ele tentará abrir um stream para leitura de caracteres (2), tentará ler o arquivo em blocos e imprimir o que foi lido. Se tudo funcionar sem erros (3), o bloco *finally* é executado, onde o stream é fechado (5), e o método continua (6).
- **Exceção capturada.** Após o início do *try* (2), se houver algum erro no bloco *try* (por exemplo, o arquivo poderá não ter permissão para leitura), uma exceção será criada e lançada, e o controle será interrompido. O bloco *catch* (3) simplesmente informa que o arquivo não foi lido. Depois, o stream é fechado, se ele chegou a ser aberto (5), no *finally*, e o método continua (6).
- **Exceção não capturada.** Depois de iniciado o *try* (2) pode também ocorrer um erro do sistema ou alguma exceção de runtime que não foi prevista. Como o *catch* não captura a exceção, ela não será tratada, mas o *finally* ainda assim será executado e se o stream chegou a ser aberto, ele será fechado (5), mas o método irá retornar, e não continuará.

### 2.2.3 Try com resources

Recursos como streams e banco de dados que são limitados e precisam ser liberados sempre têm um método *close()* que deve ser chamado no *finally*, mas é possível também declarar um bloco *try* de tal forma que o fechamento seja automático, inicializando os recursos que precisam ser fechados dentro de parênteses depois do *try*:

```

try (FileReader input = new FileReader("texto.txt"))
{
    System.out.println("2) Tentativa de ler o arquivo");
    byte[] buffer = new byte[1024];
    int len = input.read(buffer);
    while(len != -1){
        System.out.write(buffer, 0, len);
        len = input.read(buffer);
    }
    System.out.flush();
    System.out.println("3) Arquivo lido com sucesso");
} catch (IOException e) {
    System.out.println("4) Erro de IO - arquivo não foi lido!");
}

```

O *finally* pode ainda ser necessário para outras tarefas, e talvez não seja removido.

Se houver mais recursos a fechar, inicializações adicionais poderão ser incluídas nos parênteses do try desde que separadas por parênteses.

### 3 Como declarar

Exceções podem ser declaradas em métodos e construtores. Exceções devem ser capturadas e tratadas o mais próximo possível do local em que ocorrem. Mas às vezes um método não tem como tratar o problema eficientemente, e retorna para chamador. Neste caso, deve declarar que pode lançar a exceção usando a cláusula *throws*:

```
public void método() throws IOException, FileNotFoundException { ... }
```

A cláusula *throws* é muito importante como documentação. Através dela é possível saber como lidar com a potencial exceção que será lançada pelo método. Se um método provoca várias exceções que fazem parte de uma hierarquia, o ideal é que declare provocar as exceções separadas em vez de uma única exceção (a superclasse), pois isso dá mais opções a quem vai usar o método. Como *FileNotFoundException* é subclasse de *IOException*, esta declaração poderia ser usada em vez na mostrada anteriormente, mas é menos abrangente:

```
public void método() throws IOException { ... }
```

Exceções *não-checadas* (como *NumberFormatException*) não devem ser declaradas, mas podem e devem ser documentadas em comentários Javadoc.

```

/** Abre um arquivo.
 * @throws NumberFormatException se o código
 *         contido no arquivo não for um inteiro.
 */
void open(File arquivo) throws FileNotFoundException{ ... }

```

Em uma sobreposição, o método novo não pode declarar mais exceções que o método original. Isto pode forçar a captura de exceções dentro desses métodos, mesmo que para lançar dentro de outra. Se na superclasse o método abaixo causa apenas *FileNotFoundException*:

```
public abstract void process(File f) throws FileNotFoundException, MyAppException;
```

Na subclasse, a sobreposição não pode causar *IOException* (que abrange mais que *FileNotFoundException*), mas se ele declara uma exceção geral própria adequada a informar outros erros (*ServletException*), podemos enviar outras exceções dentro dela:

```

@Override public void process(File f) throws FileNotFoundException, ServletException {
    try {
        metodoLeitura(); // causa IOException
    } catch (IOException e) {
        throw new ServletException(e); // encapsulando IOException em ServletException
    } ...
}

```

Às vezes uma exceção é capturada e relançada. Isto pode acontecer por vários motivos. Por exemplo, para que o método acrescente dados na exceção, ou porque ele não é capaz de solucionar o problema totalmente, e pode resolver parte dele, ou se ele desejar logar a exceção, etc.:

```
@Override
public void process(File f) throws IOException {
    try {
        metodo(); // causa IOException
    } catch (IOException e) {
        if(e instanceof FileNotFoundException) {
            f = new File("backup.txt");
        } else {
            throw e; // relança a exceção
        }
    }
    ...
}
```

## 4 Como criar exceções

Uma exceção pode ser criada *estendendo* a classe *Exception*. Uma exceção pode ter atributos, métodos, se necessário. Geralmente não tem nada, apenas construtores. É comum simplesmente implementar os construtores da superclasse. Pode-se criar uma exceção apenas com o construtor *default*. Usar o sufixo *Exception* é uma boa prática:

```
public class SemEspacoDeArmazenamentoException extends Exception {}
```

Agora é possível usar a exceção em cláusulas *catch*, declarações *throws* e instruções *throw*:

```
public void m() throws SemEspacoDeArmazenamentoException {
    //...
    if(erro) {
        throw new SemEspacoDeArmazenamentoException();
    }
    ...
}
```

A classe *Exception* possui construtores que permitem passar uma mensagem em forma de *String* e *encadear* exceções incluindo uma referência para outra exceção (como foi feito com *ServletException* em exemplo anterior). Portanto, é uma boa prática criar exceções que também implementem esses construtores (que podem ser gerados pelo Eclipse):

```
public class SemEspacoDeArmazenamentoException extends Exception {

    public SemEspacoDeArmazenamentoException() {}

    public SemEspacoDeArmazenamentoException(String message) {
        super(message);
    }

    public SemEspacoDeArmazenamentoException(Throwable cause) {
        super(cause);
    }

    public SemEspacoDeArmazenamentoException(String message, Throwable cause) {
        super(message, cause);
    }

    public SemEspacoDeArmazenamentoException(String message, Throwable cause,
                                              boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

Agora é possível informar os problemas de forma mais clara:

```
try {
    método();
} catch (IOException e) {
    throw new SemEspacoDeArmazenamentoException("Não pode gravar no disco", e);
}
```

Na cláusula *throw* acima, a exceção foi *vinculada* com o *IOException* que a causou.

## 5 Reuso de exceções

É importante conhecer as classes de exceções, não apenas para facilitar a depuração, mas também para saber quando usá-las. Em geral é melhor reusar uma exceção do que criar uma classe nova. Se você quer tratar a condição excepcional de receber argumentos inválidos para um método, em vez de criar uma nova *ArgumentosInvalidosException*, você pode reusar a *IllegalArgumentException*, que foi criada especificamente para este fim. Portanto, evite criar exceções sem necessidade. Prefira reusar.

### 5.1 Algumas exceções que devem ser reusadas

- **IllegalStateException** – use para sinalizar que uma chamada é ilegal devido ao estado do objeto (por exemplo, chamar um objeto ainda não inicializado)
- **IllegalArgumentException** – use para acusar valor inadequado para parâmetro de um método ou construtor (ex: valor negativo para um contador)
- **IndexOutOfBoundsException** - Se o cliente passa um valor fora de escopo
- **NullPointerException** - Se o cliente passa null quando valores nulos são proibidos
- **UnsupportedOperationException** – use quando uma implementação não implementa parte da interface

Reuse as exceções de APIs padrão: *IOException*, *SQLException*, *RemoteException*, etc. Sempre que possível, seja mais específico. Ex: *FileNotFoundException* em vez de *IOException*, quando for o caso.

Isto vale também para as exceções padrão de frameworks, como Java EE: *ServletException*, *EJBException*, etc.

# 7 Threads

<b>1 Multithreading em Java</b>	<b>1</b>
<b>1.1 Criação de threads</b>	<b>1</b>
1.1.1 A Interface Runnable	1
1.1.2 A classe Thread	2
1.1.3 Criação de threads sem usar Runnable	3
<b>1.2 Métodos de Thread</b>	<b>4</b>
<b>1.3 Ciclo de vida</b>	<b>4</b>
<b>1.4 Controle de threads</b>	<b>5</b>
<b>1.5 Sincronização</b>	<b>5</b>
<b>1.6 Sintaxe usando classes internas</b>	<b>6</b>

Threads são abstrações que representam e implementam paralelismo em Java. A capacidade de executar tarefas em paralelo é fundamental para construir aplicações interativas, servidores, etc.

Existem várias maneiras de usar threads em Java. O mais comum é não usar, e deixar a responsabilidade para um framework que se encarregue de criar os threads e sincronizá-los. *Threads* são usados automaticamente na API nativa de interface gráfica e nos frameworks do Java EE. Porém, para trabalhar com as APIs de baixo nível de rede e IO, o programador precisa criar e sincronizar seus próprios threads.

Existe uma API básica de baixo nível baseada na classe *Thread* e na interface *Runnable*, que oferece a infraestrutura básica. Para aplicações mais complexas, o ideal é usar a API de Executores do pacote *java.io.concurrent*. Nesta seção apresentaremos uma breve introdução com os conceitos essenciais de multithreading com a API básica.

## 1 Multithreading em Java

Uma aplicação em Java possui pelo menos um thread de execução, que executa o método `main()`. Usando implementações da interface *Runnable* é possível construir objetos que poderão ser passados para objetos da classe *Thread*, e iniciados em threads paralelos.

### 1.1 Criação de threads

É muito fácil criar novos threads em Java. Não é tão fácil controlá-los. Quase tudo pode ser realizado usando a interface *Runnable* e a classe *Thread*.

#### 1.1.1 A Interface Runnable

O código que irá rodar em um thread deve ser uma implementação da interface *Runnable*. *Runnable* é uma interface funcional que possui um único método:

```
public void run();
```

A implementação de Runnable deve implementar o código que irá rodar em paralelo no seu método run(). No exemplo abaixo o método run() irá contar um carneirinho por tempo de espera em milissegundos, até que a contagem chegue a 60:

```
public class Paralelo implements Runnable {
    public volatile int contagem;

    private int espera;
    private String tipo;

    public Paralelo(String tipo, int espera) {
        this.tipo = tipo;
        this.espera = espera;
    }

    public void run() {
        while (contagem < 60) {
            System.out.println(++contagem + " carneirinhos " + tipo + "...");
            try { Thread.sleep(espera); } catch (InterruptedException e) {}
        }
        System.out.println("Não há mais carneirinhos " + tipo);
    }
}
```

### 1.1.2 A classe Thread

Para criar um thread é preciso instanciar o objeto Runnable e passa-lo como argumento de uma instância da classe Thread, que funciona como uma CPU virtual que irá rodar o código de Runnable dentro de uma linha de execução paralela:

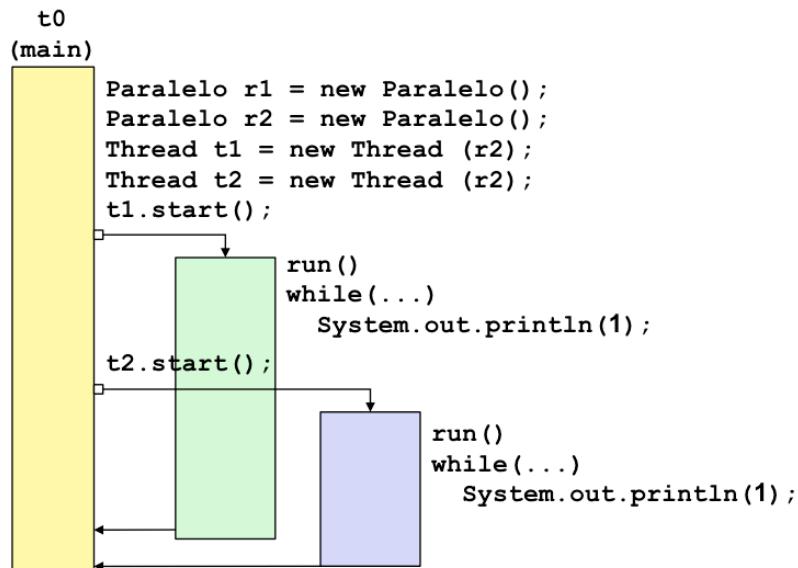
```
Paralelo r1 = new Paralelo("da montanha", 1000);
Thread carneiros1 = new Thread(r1);

Paralelo r2 = new Paralelo("do campo", 500);
Thread carneiros2 = new Thread(r2);
```

Depois de criado, o objeto pode ainda ser configurado se necessário e, quando for a hora de executar, chamar start():

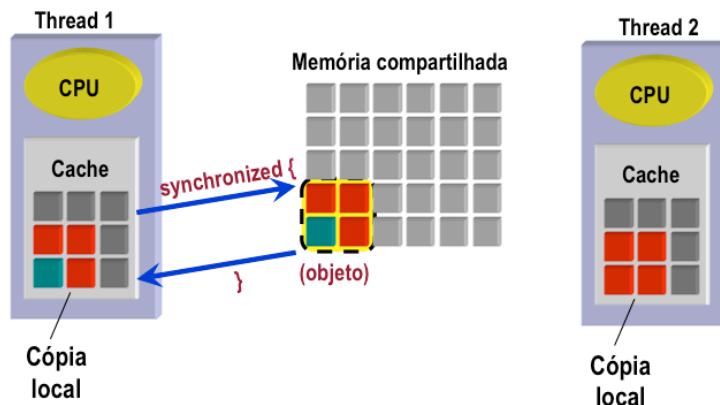
```
carneiros1.start();
carneiros2.start();
```

Depois do start, cada objeto roda em um thread paralelo:



O thread acima opera em cima de dados locais a cada instância, mas a variável *contagem* é pública, e poderia ser alterada por outro Thread. Ela foi marcada como *volatile* para garantir que se outro thread alterá-la, o valor dela seja automaticamente sincronizado (o while, que mantém a execução do thread, depende disso). Para métodos e blocos de código usa-se *synchronized* para obter esse efeito.

O diagrama abaixo ilustra uma máquina rodando dois threads em CPUs diferentes. Cada CPU retém um cache local da memória. Os modificadores *synchronized* ou *volatile* garantem que os dados locais estarão sempre sincronizados (essencial para que os outros threads vejam e possa reagir às suas alterações de valor).



### 1.1.3 Criação de threads sem usar Runnable

Uma outra maneira de criar e iniciar um Thread é estendendo diretamente a classe Thread (que também implementa Runnable):

```
public class ParaleloThread extends Thread {
    public volatile int contagem;

    private int espera;
    private String tipo;

    public ParaleloThread(String tipo, int espera) {
        this.tipo = tipo;
        this.espera = espera;
    }
    public void run() {
        while (contagem < 60) {
            System.out.println(++contagem + " carneirinhos " + tipo + "...");
            try {
                Thread.sleep(espera);
            } catch (InterruptedException e) {} // espera 1 segundo
        }
        System.out.println("Não há mais carneirinhos " + tipo);
    }
}
```

Para criar o objeto, neste caso, usamos um passo a menos:

```
Paralelo t = new Paralelo("do campo", 500);
t.start();
```

## 1.2 Métodos de Thread

Uma vez criado um objeto Thread, ele pode ser manipulado através de diversos métodos e funções estáticas. Os principais métodos de instância utilizáveis por objetos da classe Thread são:

- **start()**: inicia uma nova linha de controle com base nos dados do objeto Thread e invoca o seu método run();
- **getId()**: retorna o id do thread (um número long gerado automaticamente)
- **getName()**: retorna o nome do thread (pode ser definido com setName())
- **isAlive()**: retorna true se o thread estiver ativo
- **join()**: faz com que o thread que chamou o método espere que o thread no qual join() é chamado, termine para continuar.

Os principais métodos estáticos da classe Thread são:

- **sleep(milisegundos)**: suspende a execução do thread ativo neste momento por um determinado período de no mínimo milisegundos.
- **yield()**: dá a preferência a outras linhas que estiverem esperando pela oportunidade de executar.
- **currentThread()**: retorna a instância de Thread que está executando no momento.

Estes métodos podem alterar o *estado* de um thread.

## 1.3 Ciclo de vida

Um Thread têm um ciclo de vida que inicia quando é criado e termina quando morre. Durante a sua vida, um *thread* pode existir em três estados:

- *Executando*, quando ele consegue espaço da CPU para realizar o que foi programado para fazer
- *Esperando*, quando está inativo por alguma razão (suspenso, dormindo, aguardando notificação, bloqueado).
- *Pronto (ready)*, quando não espera nada a não ser a oportunidade de executar.

O quarto estado é *morto*, que ocorre quando um thread termina.

A maneira como um thread se comporta é dependente de plataforma. No exemplo que criamos, forçamos um compartilhamento de tempo fazendo com que os threads dormissem por um tempo. Desta maneira, um *thread* está sempre alternando com os outros *threads* entre o estado *pronto* e o estado *executando*.

Vários *threads* podem estar ao mesmo tempo no estado *pronto*, esperando sua vez na CPU. Algumas plataformas determinam compartilhamento de tempo automaticamente. Outras alocam o thread com base em prioridades. Se houver um *thread* de maior prioridade na sua frente, o thread terá que esperar. Se não houver prioridade, ele não pode ter certeza quando vai ser chamado. Não há garantia que o *thread* que espera há mais tempo será chamado.

O método setPriority() pode ser chamado para definir a prioridade de threads. Os valores passados podem ser qualquer valor de 1 a 10.

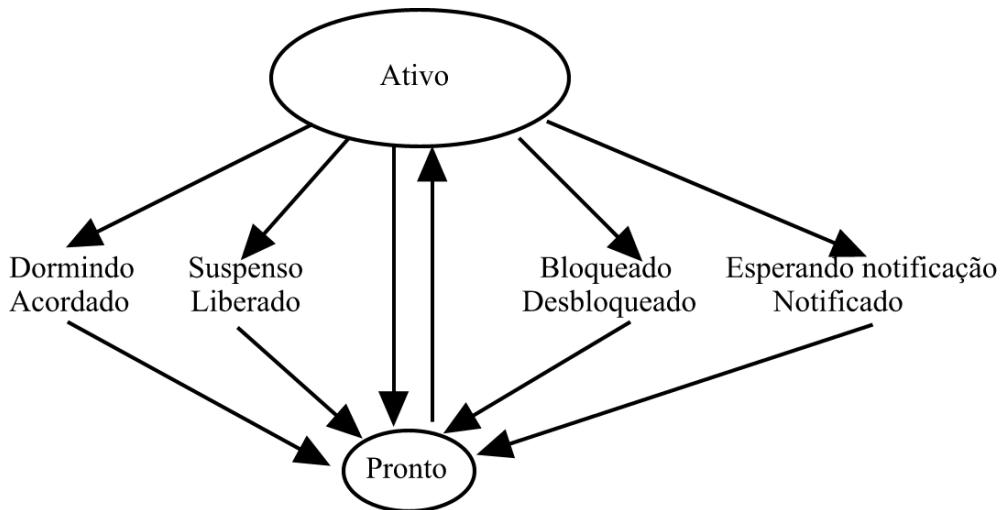
Como o funcionamento de prioridades em *threads* é dependente de plataforma, algoritmos que dependem das prioridades dos *threads* poderão gerar resultados imprevisíveis.

## 1.4 Controle de threads

Há várias maneiras de controlar a mudança de estado de um *Thread*:

- Dando a preferência (chamando a função `yield()`)
- Dormindo e acordando (chamando a função `sleep(tempo)`)
- Bloqueando (efeitos externos diversos e interrupções)
- Esperando (`wait()`) e sendo notificado (`notify()`)

A figura abaixo ilustra os estados de um *thread* vivo.



O método `yield()` é estático. Opera no thread ativo (`currentThread`). É usado pelo thread que ocupa a CPU quando quer dar uma oportunidade aos outros threads. `yield()` faz com que ele deixe a CPU e vá para o estado pronto. Se houver algum thread interessado na CPU, este ganha a vez. Caso contrário volta para a CPU.

O método `sleep()` coloca um thread para dormir durante um determinado tempo. Também opera no `currentThread`. Depois que um thread acorda, ele não volta a ocupar a CPU imediatamente, mas muda para o estado pronto e aguarda a sua vez.

O bloqueio ocorre quando um fator externo ou uma interrupção (usando `interrupt()`) causa a espera do thread. Por exemplo, uma operação de leitura em disco. Quando a causa do bloqueio se vai, novamente o thread entra no estado de prontidão e espera sua vez pela CPU.

## 1.5 Sincronização

Java usa a palavra-chave `synchronized` para declarar blocos de código, métodos e classes que devem ter os dados em que operam bloqueados contra acessos simultâneos e a possível corrupção de dados.

Normalmente `synchronized` é usado como modificador de um método, embora possa ser usado para isolar pequenos blocos de código dentro de um método:

Se um thread chama um método declarado com o modificador `synchronized` o objeto no qual o método reside é bloqueado para outros threads. A sincronização garante que a execução de suas linhas serão mutualmente exclusivas no tempo. Os métodos sincronizados esperam enquanto um outro termina seu trabalho antes de operar no mesmo conjunto de dados.

Os métodos de `Object` `wait()`, `notify()` e `notifyAll()` podem ser usados para monitorar threads e só podem ocorrer dentro de blocos `synchronized`.

## 1.6 Sintaxe usando classes internas

Muitas vezes os métodos run() são curtos e threads são criados e iniciados em poucas linhas. Um padrão comum é usar a sintaxe de classes internas ou lambdas, como no exemplo abaixo:

```
new Thread( new Runnable() {
    public void run() {
        while(carneiros1.isAlive() || carneiros2.isAlive()) {
            int sorte = (int)(Math.random() * 50);
            if (sorte == 25) {
                System.out.println(">>> Redefinindo contagem!");
                r1.contagem = 50;
                r2.contagem = 25;
            }
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {} // espera 1 segundo
        }
        System.out.println("No threads alive!");
    }
}).start();
```

# 8 Utilitários

---

<b>1 Datas</b>	<b>1</b>
<b>1.1 Benchmarking</b>	1
<b>1.2 Date</b>	2
<b>1.3 Calendar</b>	2
1.3.1 GregorianCalendar	2
<b>1.4 Pacote java.time</b>	3
<b>2 Internacionalização</b>	<b>3</b>
<b>2.1 Locale</b>	3
<b>2.2 DateFormat</b>	3
2.2.1 SimpleDateFormat	4
<b>2.3 NumberFormat</b>	4
2.3.1 DecimalFormat	4
<b>3 Matemática</b>	<b>5</b>
<b>3.1 Arredondamento</b>	5
<b>3.2 Geração de números pseudo-aleatórios</b>	6

## 1 Datas

Uma representação default de uma data é proporcionada pela classe *java.util.Date*, que armazena um instante de tempo em um long. Uma data de calendário, com mês, dia, dia da semana, hora, minuto e segundo é representada pela classe *Calendar*, que implementa o calendário ocidental na classe *GregorianCalendar*, que pode ser usado para construir uma representação de data.

### 1.1 Benchmarking

O método *System.currentTimeMillis()* retorna o instante atual em milissegundos. É um método prático para construir benchmarks (registrar dois instantes e calcular o tempo transcorrido). É equivalente a new *Date().getTime()*.

```
long inicio = System.currentTimeMillis();
// realiza várias tarefas
long tempoTranscorrido = System.currentTimeMillis() - inicio;
```

## 1.2 Date

A classe *Date* representa tempo com precisão de milissegundos. Não é a classe ideal para construir datas usando calendários, mas talvez seja a mais comum. Para criar um objeto *Date()* representando o instante atual:

```
Date d = new Date();
```

Para obter o tempo UTC (contagem de milissegundos desde 1970):

```
Long utc = d.getTime();
```

Uma data pode ser construída no passado ou futuro usando UTC:

```
Date amanha = new Date(utc + 86400000);
```

Se impresso como String, *Date* imprime a representação completa da data e hora, com fuso horário, e de acordo com o Locale default. O formato pode ser alterado usando um *DateFormat* ou *SimpleDateFormat*.

Para manipular com meses, dias, horas, anos, etc, deve-se usar *Calendar*.

## 1.3 Calendar

*Calendar* é uma classe abstrata que representa um calendário que consiste de atributos tais como mês, ano, dia da semana, etc. Contém constantes que representam esses campos e vários métodos para extraí-los de uma representação de calendário.

*Calendar* é sensível ao locale. O instante atual, no locale atual pode ser obtido usando:

```
Calendar agora = Calendar.getInstance();
```

Que é similar a *new Date()*.

Uma vez criado um *Calendar*, os campos desejados podem ser lidos usando o método *get(campo)*:

```
System.out.println("ERA: " + agora.get(Calendar.ERA));
System.out.println("YEAR: " + agora.get(Calendar.YEAR));
System.out.println("MONTH: " + agora.get(Calendar.MONTH));
System.out.println("WEEK_OF_YEAR: " + agora.get(Calendar.WEEK_OF_YEAR));
System.out.println("WEEK_OF_MONTH: " + agora.get(Calendar.WEEK_OF_MONTH));
System.out.println("DATE: " + agora.get(Calendar.DATE));
System.out.println("DAY_OF_MONTH: " + agora.get(Calendar.DAY_OF_MONTH));
System.out.println("DAY_OF_YEAR: " + agora.get(Calendar.DAY_OF_YEAR));
System.out.println("DAY_OF_WEEK: " + agora.get(Calendar.DAY_OF_WEEK));
System.out.println("DAY_OF_WEEK_IN_MONTH:" + agora.get(Calendar.DAY_OF_WEEK_IN_MONTH));
System.out.println("AM_PM: " + agora.get(Calendar.AM_PM)); System.out.println("HOUR: " +
+ agora.get(Calendar.HOUR)); System.out.println("HOUR_OF_DAY: " +
+ agora.get(Calendar.HOUR_OF_DAY)); System.out.println("MINUTE: " +
+ agora.get(Calendar.MINUTE)); System.out.println("SECOND: " +
+ agora.get(Calendar.SECOND)); System.out.println("MILLISECOND: " +
+ agora.get(Calendar.MILLISECOND));
```

### 1.3.1 GregorianCalendar

Implementa a classe abstrata *Calendar* com o calendário ocidental. É o calendário default para a maior parte dos locales. Esta instrução cria um *Calendar* para o instante atual (mesmo que *new Date()*):

```
Calendar calendar = new GregorianCalendar();
```

## 1.4 Pacote `java.time`

A partir do Java 8 foi introduzido o pacote `java.time`, que contém as principais abstrações para instantes, durações, datas, períodos, fusos-horários, etc.

As classes mais importantes são `Instant` (que é equivalente a `java.util.Date`), `ZonedDateTime` (equivalente a `java.util.Calendar`), `LocalDate`, `LocalTime` e `LocalDateTime`. É uma API muito mais precisa e organizada que todas as anteriores, mas como é muito nova, é usada apenas nas APIs e frameworks mais recentes.

## 2 Internacionalização

O pacote `java.text` possui diversas classes que lidam com formatação de texto, acentuação, encoding, locales, formatação de datas, moeda, números, etc. Algumas delas são apresentadas nesta seção.

### 2.1 Locale

Um objeto `Locale` representa uma região geográfica, política ou cultural. Mostrar uma data ou número são operações sensíveis ao `Locale` – a data ou número devem ser formatados de acordo com os costumes e convenções do país, região ou cultura.

Um `Locale` consiste de vários códigos internacionais, dentre eles códigos de idioma (“en”, “pt”, “es”), alfabeto (“Latn”, “Cyrl”) e região (“BR”, “US”, “UK”). Tipicamente um `Locale` é construído a partir de um idioma ou idioma e região:

```
Locale loc = new Locale("pt");
Locale loc = new Locale("pt", "BR");
```

`Locales` são usados em várias classes que realizam formatação de informações sensíveis a `Locale`, como formatação de datas e moedas.

### 2.2 DateFormat

A classe `DateFormat` é um objeto usado para formatar datas. Para criar um é preciso selecionar um estilo de formatação e passar para o método de fábrica que cria o objeto:

```
DateFormat df = DateFormat.getDateInstance(estilo, locale);
DateFormat dtf = DateFormat.getTimeInstance(estilo, locale);
```

O estilo pode ser

- `DateFormat.LONG`
- `DateFormat.SHORT`
- `DateFormat.LONG`
- `DateFormat.FULL`

A data será formatada de acordo com as convenções do `Locale` usado. Para formatar uma data existente (converter a data em texto):

```
Date data = new Date();
String texto = df.format(data);
```

Para processar (parse) um texto no estilo especificado e convertê-lo em data:

```
Date data = df.parse(texto);
```

### 2.2.1 SimpleDateFormat

A classe *SimpleDateFormat* é uma subclasse de *DateFormat*, e também realiza a conversão Data->Texto e Texto->Data, mas em vez de usar estilos prontos, permite que a data seja formatada usando *um padrão*. Este padrão usa letras e símbolos para determinar como a data será formatada ou processada e está detalhado na documentação oficial.

Para criar:

```
SimpleDateFormat sdf =
    new SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss Z", new Locale("pt","BR"));
```

Para usar:

```
String texto = sdf.format(data);
Date data = sdf.parse(texto);
```

A tabela abaixo (da documentação oficial) mostra alguns padrões que podem ser passados para o construtor de um *SimpleDateFormat*, e seu resultado:

Padrão	Resultado
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o'clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSXXX"	2001-07-04T12:08:56.235-07:00
"YYYY- 'W'ww-u"	2001-W27-3

### 2.3 NumberFormat

*NumberFormat* permite formatar um número para um locale específico. Pode-se obter um *NumberFormat* usando:

```
NumberFormat nf = NumberFormat.getInstance(new Locale("pt","BR"));
```

E depois usa-lo para formatar números de acordo com esse Locale:

```
double numero = 34.52;
String numeroStr = nf.format(numero); // 34,52 (troca ponto pela vírgula)
```

Pode-se usar *NumberFormat* para converter representações numéricas em números:

```
Double numero = nf.parse(numeroStr);
```

### 2.3.1 DecimalFormat

*DecimalFormat* é uma subclasse de *NumberFormat* que, de forma análoga a *SimpleDateFormat*, oferece um padrão (ex: "#,##0.00") que pode ser usado para determinar a base para converter Número em texto e vice-versa.

```
DecimalFormat dcf = new DecimalFormat("#,##0.00");
String numeroStr = dcf.format(numero);
Double numero = dcf.parse(numeroStr);
```

### 3 Matemática

Java disponibiliza através da classe `java.lang.Math`, uma coleção de constantes e funções básicas de matemática e trigonometria. A lista abaixo contém algumas delas:

- static final double **E** – constante  $e$
- static final double **PI** – constante  $\pi$
- static double **atan**(double a) – arco tangente
- static double **acos**(double a) – arco cosseno
- static double **asin**(double a) – arco seno
- static double **tan**(double a) - tangente
- static double **cos**(double a) - cosseno
- static double **sin**(double a) - seno
- static double **tanh**(double x) – tangente hiperbólica
- static double **cosh**(double x) – cosseno hiperbólico
- static double **sinh**(double x) – seno hiperbólico
- static double **exp**(double a) –  $e$  elevado à potência de a
- static double **log**(double a) – logaritmo natural
- static double **log10**(double a) – logaritmo de base 10
- static double **sqrt**(double a) – raiz quadrada
- static double **cbrt**(double a) – raiz cúbica
- static double **ceil**(double a) – arredonda para cima
- static double **floor**(double a) – arredonda para baixo
- static int **round**(double a) – arredonda para cima ou para baixo
- static double **pow**(double a, double b) – a elevado à potência de b
- static double **random()** – retorna um número pseudo aleatório entre 0 e 1
- static int **abs**(double a) - absoluto
- static double **max**(double a, double b) – retorna o maior dos dois
- static double **min**(double a, double b) – retorna o menor dos dois
- static double **hypot**(double x, double y) - hipotenusa
- static double **toRadians**(double angdeg) – converte para radianos
- static double **toDegrees**(double angrad) – converte para graus

#### 3.1 Arredondamento

`Math.floor()` trunca a parte decimal, `Math.ceil()` arredonda para cima, e `Math.round()` arredonda para cima ou para baixo dependendo do valor. O valor retornado por `round()` é `long`, mas `floor()` e `ceil()` retornam `double`, portanto pode ser necessário converter o resultado para `int` ou `long`:

```
double numero1 = 123.56;
double numero2 = 123.46;
System.out.println("floor("+numero1+"): " + Math.floor(numero1));
System.out.println("ceil("+numero1+"): " + Math.ceil(numero1));
System.out.println("round("+numero1+"): " + Math.round(numero1));
System.out.println("floor("+numero2+"): " + Math.floor(numero2));
System.out.println("ceil("+numero2+"): " + Math.ceil(numero2));
System.out.println("round("+numero2+"): " + Math.round(numero2));
```

Imprime:

```
floor(123.56): 123.0
ceil(123.56): 124.0
round(123.56): 124
floor(123.46): 123.0
ceil(123.46): 124.0
round(123.46): 123
```

### 3.2 Geração de números pseudo-aleatórios

Para gerar um número pseudo-aleatório pode-se usar *Math.random()*, que retorna um *double* entre 0 e 1. O número pode ser multiplicado e truncado para produzir um resultado inteiro:

```
System.out.println("(int)(Math.random() * 100): " + (int)(Math.random() * 100));
```

Se for necessário mais controle, pode-se usar a classe *Random* que tem métodos para gerar doubles (entre 0 e 1), ints (dentro de um limite estabelecido), booleans e ainda streams de números pseudo-aleatórios. Também permite alterar a semente usada na geração.

Exemplo de uso:

```
Random num = new Random();
num.setSeed(new Date().getTime());
System.out.println("nextInt(3): " + num.nextInt(3));
System.out.println("nextDouble(): " + num.nextDouble());
```

# 9 Arquivos e I/O

---

<b>1 Representação de arquivos</b>	<b>1</b>
<b>2 Data streams</b>	<b>2</b>
<b>2.1 Leitura de dados e caracteres</b>	<b>2</b>
<b>2.2 Gravação de dados e caracteres</b>	<b>4</b>
<b>2.3 Concatenação de streams usando decoradores</b>	<b>5</b>
<b>2.4 IOException</b>	<b>7</b>
<b>3 Serialização de objetos</b>	<b>7</b>
<b>4 Entrada e Saída Padrão</b>	<b>8</b>
<b>5 Comunicação em rede TCP/IP</b>	<b>9</b>
<b>5.1 URLs e endereços Internet</b>	<b>9</b>
<b>5.2 Soquetes TCP/IP</b>	<b>9</b>
<b>6 Non-blocking IO (java.nio)</b>	<b>10</b>

O pacote `java.io` inclui classes e interfaces relacionadas à transferência de dados de e para aplicações Java, e abstrações que representam arquivos e diretórios em um sistema local.

## 1 Representação de arquivos

O pacote `java.io` contém ainda duas classes que representam arquivos em disco. `File` e `RandomAccessFile`. A primeira, usada nos exemplos apresentados, descreve um arquivo e possui vários métodos para manipular com eles.

Os métodos de `File` não criam arquivos comuns (isto requer um `FileWriter` ou `FileOutputStream`), mas podem removê-los, verificar se existem, criar pastas e listar seu conteúdo. `File` lida com arquivos de forma independente de plataforma, automaticamente convertendo as representações locais (ex: separadores / ou \).

Pode-se criar um `File` passando como argumento um caminho absoluto (ou URI) para um arquivo (que não precisa existir), ou um caminho relativo a um diretório passado como primeiro argumento. `File` pode representar um arquivo comum ou um diretório:

```
File inbox = new File("/users/root/inbox");
File mensagem = new File(inbox, "mensagem.txt");
```

Para criar um arquivo comum é preciso gravar alguma coisa através de um `FileOutputStream` ou `FileWriter`. Diretórios, porém, podem ser criados diretamente usando o método `mkdir()`:

```
inbox.mkdir();      // cria diretório se possível
FileWriter out = new FileWriter(mensagem);
out.write("teste"); // se arquivo não existe, tenta criar
```

Outros métodos de File realizam testes, obtém informações e alteram arquivos existentes. A seguir, os principais construtores e métodos. Os nomes dos métodos são bastante auto-explicativos, portanto, não detalharemos seu funcionamento.

Construtores:

```
public File(String caminho);
public File(String caminho, String nome);
public File(File diretorio, String caminho);
```

Principais Métodos:

```
public boolean exists();
public boolean delete();
public boolean isDirectory();
public boolean isFile();
public boolean mkdir();
public boolean mkdirs();
public boolean renameTo(File novo_arquivo);
public boolean equals(Object obj);

public String getAbsolutePath();
public String getName();
public String getPath();
public String[] list();
public String[] list(FilenameFilter filtro);

public long lastModified();
public long length();
```

## 2 Data streams

O processo de entrada/saída (input/output) é a transferência de dados entre uma aplicação e um sistema externo. Em Java isto é realizado através de *fluxos de dados*, ou *data streams*.

Existem dois tipos:

- *Fluxos de entrada*, ou seja, da fonte externa para a aplicação, e
- *Fluxos de saída*, que corresponde à transferência da aplicação para um destino externo.

Esses fluxos são representados por dois pares de interfaces:

- *java.io.InputStream* e *java.io.OutputStream* – fluxos de entrada e saída de bytes
- *java.io.Reader* e *java.io.Writer* – fluxos de entrada e saída de caracteres Unicode

Um stream de dados ou caracteres de entrada é criado a partir da leitura sequencial das informações de uma fonte de dados, que podem ter origem em na memória, em arquivos locais ou mesmo uma conexão de rede.

### 2.1 Leitura de dados e caracteres

As interfaces *InputStream* e *Reader* declaram métodos de leitura que precisam ser implementados pelas classes que geram streams dessas fontes. Estes são alguns dos principais métodos de um *InputStream*:

- *int read()* – lê um único byte do stream
- *int read(byte[] buffer)* – preenche um array com bytes lidos

- *int read(byte buffer, int offset, int length)* – preenche parte do array com bytes lidos
- *abstract void close()* – fecha o stream

E de um *Reader*:

- *int read()* – lê um único caractere do stream
- *int read(char[] buffer)* – preenche um array com chars lidos
- *int read(char buffer, int offset, int length)* – preenche parte do array com chars lidos
- *abstract void close()* – fecha o stream

Alguns outros métodos permitem estimar quantos bytes/caracteres ainda há para ler e usar mecanismos para marcar e pular caracteres e bytes. Abaixo algumas implementações de implementações que geram streams de entrada:

Tipo de fonte	Leitura de bytes	Leitura de caracteres
Arquivo local	<code>FileInputStream</code>	<code>FileReader</code>
Memória	<code>ByteArrayInputStream</code>	<code>CharArrayReader</code> <code>StringReader</code>
Dutos (Pipes)	<code>PipedInputStream</code>	<code>PipedReader</code>

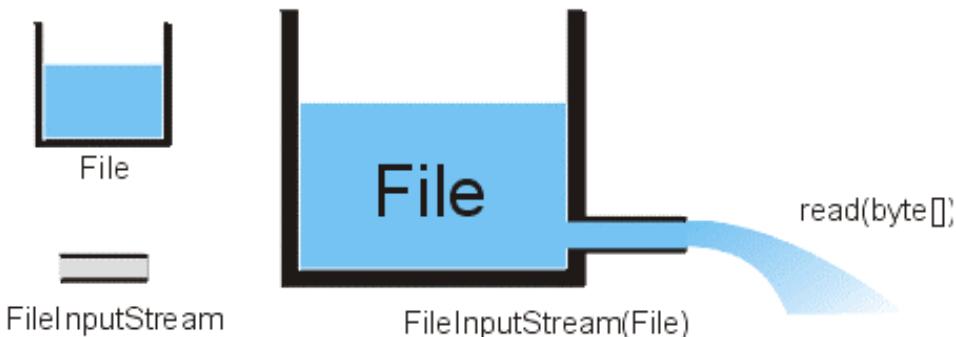
As implementações recebem os objetos que representam as fontes como parâmetro do construtor. Por exemplo, `FileInputStream` recebe um `java.io.File`, `StringReader` recebe um `String`, e assim por diante.

Para descrever o funcionamento de um fluxo de bytes pode-se usar a analogia de um fluxo de água a partir de uma caixa d'água. A caixa d'água é uma fonte de armazenamento, como um arquivo (`java.io.File`). É impraticável extrair toda a água de uma vez, mas se um cano for instalado, um fluxo de água será criado possibilitando a condução da água para outro lugar.

Uma forma de implementar o exemplo acima em Java é:

```
File tanque = new File("agua.tif");           // objeto do tipo File
FileInputStream cano =                      // referência FileInputStream
    new FileInputStream(tanque); // cano conectado no tanque
byte gota = cano.read();                     // lê um byte a partir do cano
```

Esta analogia está ilustrada no desenho abaixo:



O cano na verdade é mais como um conta-gotas, pois a leitura do arquivo não é um processo analógico. Mas ler um byte de cada vez é tão ineficiente quanto extrair água da caixa a conta-gotas. Normalmente usariamos um cano grosso capaz de encher um balde de cada vez. O balde ideal seria o maior possível (mas não grande demais; se for muito grande que a quantidade de água disponível, o transporte também será ineficiente.)

Aplicando esta analogia ao arquivo, o balde é um buffer, ou array que será usado para ler vários bytes de uma só vez. Se o array for muito grande, ele irá desperdiçar memória. Se ele for muito

pequeno, a leitura será ineficiente. Não há um tamanho ideal. É preciso ter uma noção do tamanho dos dados que serão lidos.

No exemplo abaixo temos um arquivo de pouco mais de 4kB, que requer 3 chamadas do método `read()` se o programa usado para ler seu conteúdo usar um buffer de 2k.

```
File imagem = new File("icon.jpg"); // arquivo com 4100 bytes
FileInputStream cano = new FileInputStream(imagem);

byte[] balde1 = new byte[2048]; // buffer com capacidade de 2048 bytes

cano.read(balde1); // lê 2048 bytes do arquivo
cano.read(balde1); // lê mais 2048 bytes do arquivo (4096)
cano.read(balde1); // lê os 4 bytes restantes
```

Seria um desperdício criar um buffer de 16MB:

```
byte[] balde2 = new byte[16777216]; // buffer com capacidade de 16MB
```

Que poderia ser usado para ler arquivos de 40MB. Por outro lado, com um buffer de 2k seria necessário chamar `read()` umas 20 mil vezes.

## 2.2 Gravação de dados e caracteres

Para gravar dados da aplicação para um destino, é a aplicação que precisa gerar o stream. As implementações de `OutputStream` e `Writer` são responsáveis por acumular armazenar os dados no meio de armazenamento. Os métodos da interface estão envolvidos com gravação. Estes são os métodos de um `OutputStream`:

- `void write()` – grava um único byte no stream
- `void write(byte[] buffer)` – grava um array de bytes inteiro no stream
- `void write(byte[] buffer, int offset, int length)` – grava parte de um array de bytes
- `abstract void flush()` – esvazia o stream
- `abstract void close()` – esvazia e depois fecha o stream

E de um `Writer`.

- `void write()` – grava um único caractere no stream
- `void write(char[] buffer)` – grava um array de chars inteiro no stream
- `void write(char[] buffer, int offset, int length)` – grava parte de um array de chars
- `void write(String str)` – grava um String no stream
- `void write(String str, int offset, int length)` – grava parte de um String
- `abstract void flush()` – esvazia o stream
- `abstract void close()` – esvazia e depois fecha o stream

E estas as implementações do `java.io` que consomem streams de saída:

Tipo de destino	Leitura de bytes	Leitura de caracteres
Arquivo local	<code>FileOutputStream</code>	<code>FileWriter</code>
Memória	<code>ByteArrayOutputStream</code>	<code>CharArrayWriter</code> <code>StringWriter</code>
Dutos (Pipes)	<code>PipedOutputStream</code>	<code>PipedReader</code>

Assim como na leitura, a gravação de bytes é mais eficiente se for feita em blocos. Aqui há outras questões que afetam a eficiência e que determinam o tamanho dos buffers. Gravar blocos muito

grandes é mais problemático do que lê-los. É importante cuidar para que não haja desperdício na gravação.

O exemplo abaixo mostra um processo completo de leitura e gravação de um arquivo, que funciona como uma operação de cópia. O arquivo é transferido de um lugar e gravado em outro. Na hora de gravar, é preciso saber quantos bytes foram lidos, para que não sejam gravados bytes a mais e o arquivo final fique maior que o original. Esta informação é obtida do método `read(buffer)`, que devolve um inteiro contendo o número de bytes lidos. Essa informação é usada para gravar um bloco de informações.

```
FileInputStream in = new FileInputStream(arquivoOrigem);
FileOutputStream out = new FileOutputStream(arquivoDestino);

byte[] buffer = new byte[8192];
int lidos = in.read(buffer);
while(lidos != -1) {
    out.write(buffer, 0, lidos); // le apenas bytes preenchidos do array
    out.flush(); // esvazia o buffer no arquivo
    lidos = in.read(buffer); // lê mais um buffer de bytes
}
out.close(); // fecha o fluxo de saída (arquivo liberado) - use em finally
in.close(); // fecha fluxo de entrada
```

O `while()` irá executar até que retorne `-1`, sinalizando fim de arquivo, e que não há mais bytes para ler.

## 2.3 Concatenação de streams usando decoradores

Streams podem ser concatenados. Existem vários streams que recebem como entrada um stream, devolvendo na saída outro stream, transformando os dados no processo. São chamados de decoradores, filtros ou processadores de streams e implementam a interface `FilterInputStream`. A concatenação de streams permite que os dados sejam transformados durante a leitura ou gravação.

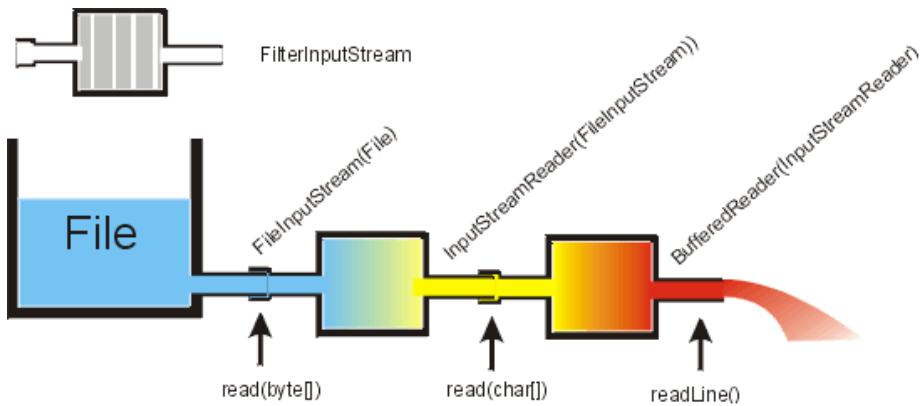
Por exemplo, existe um stream que converte `InputStreams` em `Readers`. No pacote `java.net`, usado para comunicação em TCP/IP, é possível obter um `InputStream` de um soquete, mas não um `Reader`. Se você quiser ler caracteres do soquete, pode concatenar o `InputStream` recebido a um `InputStreamReader`, que é um decorador especial que converte bytes em caracteres na leitura:

```
Socket socket = new Socket("http://localhost", 8080);
InputStream dados = socket.getInputStream();
InputStreamReader reader = new InputStreamReader(dados);
char[] buffer = new char[4096];
reader.read(buffer); // lê 4096 chars
```

Você não precisa trabalhar com `chars` e arrays de `chars`. Pode ler diretamente em `String` se concatenar na saída do `InputStreamReader` um `BufferedReader`, que possui um método `readLine()`, que permite ler os dados linha por linha:

```
InputStreamReader reader = new InputStreamReader(dados);
BufferedReader br = new BufferedReader (reader);
String linha = br.readLine(); // lê linha de texto a de br
```

O desenho abaixo ilustra a concatenação de streams:



Abaixo alguns decoradores disponíveis em `java.io`:

<code>BufferedInputStream</code> <code>BufferedOutputStream</code>	Filtro que cria automaticamente um buffer interno para otimizar a leitura e gravação de bytes no stream
<code>BufferedReader</code> <code>BufferedWriter</code>	Otimiza a leitura e gravação de streams de caracteres e Strings, oferecendo métodos para inserir novas linhas (na gravação) e ler os dados linha-por-linha (na leitura)
<code>DataInputStream</code> <code>DataOutputStream</code>	Implementam interfaces que oferecem métodos para leitura e gravação de tipos primitivos ( <code>readInt</code> , <code>readDouble</code> , etc.)
<code>LineNumberReader</code>	Stream de leitura que conta e numera linhas de texto.
<code>ObjectOutputStream</code> <code>ObjectInputStream</code>	Permitem ler e gravar objetos (serialização)

Outros pacotes que lidam com entrada e saída possuem suas próprias implementações das interfaces de streams.

Por exemplo, é possível comprimir e expandir arquivos durante a leitura e gravação usando as classes `GZipInputStream` e `GZIPOutputStream` do pacote `java.util.zip`. O exemplo abaixo ilustra o uso desses streams em combinação com serialização de objetos:

```

ObjectOutputStream out = new ObjectOutputStream(
    new java.util.zip.GZIPOutputStream(
        new FileOutputStream(arquivo)
    )
);
Objeto gravado = new Objeto();
out.writeObject(gravado);

// (...)

ObjectInputStream in = new ObjectInputStream(
    new java.util.zip.GZIPInputStream(
        new FileInputStream(arquivo)
    )
);
Objeto recuperado = (Objeto)in.readObject();

```

## 2.4 IOException

Todas as operações de entrada e saída podem não ocorrer por diversos motivos: arquivo não encontrado, falta de permissão, violação de compartilhamento, falta de memória, etc. Java trata cada um desses problemas em exceções que são subclasses de `java.io.IOException`.

A grande maioria das operações de entrada e saída provocam `IOException` ou uma das suas subexceções e o código onde são usadas deve levar isto em conta ora tratando a exceção (com `try-catch`) ou propagando-a para outros métodos (cláusula `throws`).

## 3 Serialização de objetos

Java permite que o estado de um objeto e de toda a árvore de objetos dependentes seja gravado em um array de bytes, que pode ser armazenado em disco ou enviado pela rede. Esse mecanismo é chamado de serialização e pode ser aplicada a objetos que sejam serializáveis.

Para tornar um objeto serializável é preciso declarar que sua classe implementa a interface `java.io.Serializable`, ou herdar de uma classe que implemente essa interface. Não há métodos para implementar. `Serializable` é simplesmente um flag que indica a possibilidade de converter o objeto em uma representação persistente.

```
public class Registro implements Serializable { ... }
```

Mas isto não é suficiente para que o objeto seja serializável. Ele também precisa garantir que *todas* as suas variáveis de instância, que representam o estado do objeto, *também sejam serializáveis*.

Strings e tipos primitivos são serializáveis, assim como outros objetos que declaram `Serializable`. Threads, Sockets, Streams não são e não podem ser, por sua natureza mutante. Se um objeto possuir como atributo um objeto não serializável, *ele não poderá ser serializado*, mesmo que declare `Serializable`, mas tem a opção de declarar o atributo como `transient` (e portanto não gravar o seu estado).

Por exemplo, um objeto que representa um Jogo tem um jogador, uma pontuação e uma conexão de rede (Socket). Não é possível converter uma conexão de rede em um array de bytes e armazenar seu estado. Portanto, para que seja possível gravar o estado do jogo (pontuação e usuário), conexão é declarada transient:

```
public class Jogo implements Serializable {
    private String jogador;
    private int pontuacao;
    private transient Socket conexao;
    ...
}
```

Um objeto serializável pode ser convertido em um stream usando as classes `ObjectInputStream` e `ObjectOutputStream`, que podem ser conectados a outros streams. Essas classes possuem métodos `writeObject()` e `readObject()` que recebem objetos `Serializable` como argumento.

O código abaixo ilustra como `ObjectOutputStream` e `writeObject()` podem ser usados para gravar um objeto em disco:

```
public void gravarJogo(Jogo jogo) {
    ObjectInputStream out =
        new ObjectOutputStream(
            new FileInputStream(
                new File("maria.jogo")));
    out.writeObject(jogo); // grava objeto jogo no arquivo maria.jogo
}
```

A serialização preserva o estado do objeto no momento em que é gravado. Por exemplo, se Jogo contiver `usuario="maria"` e `pontuacao=12345`, esse estado será armazenado. Mas conexão, que é *transient*, não será gravado.

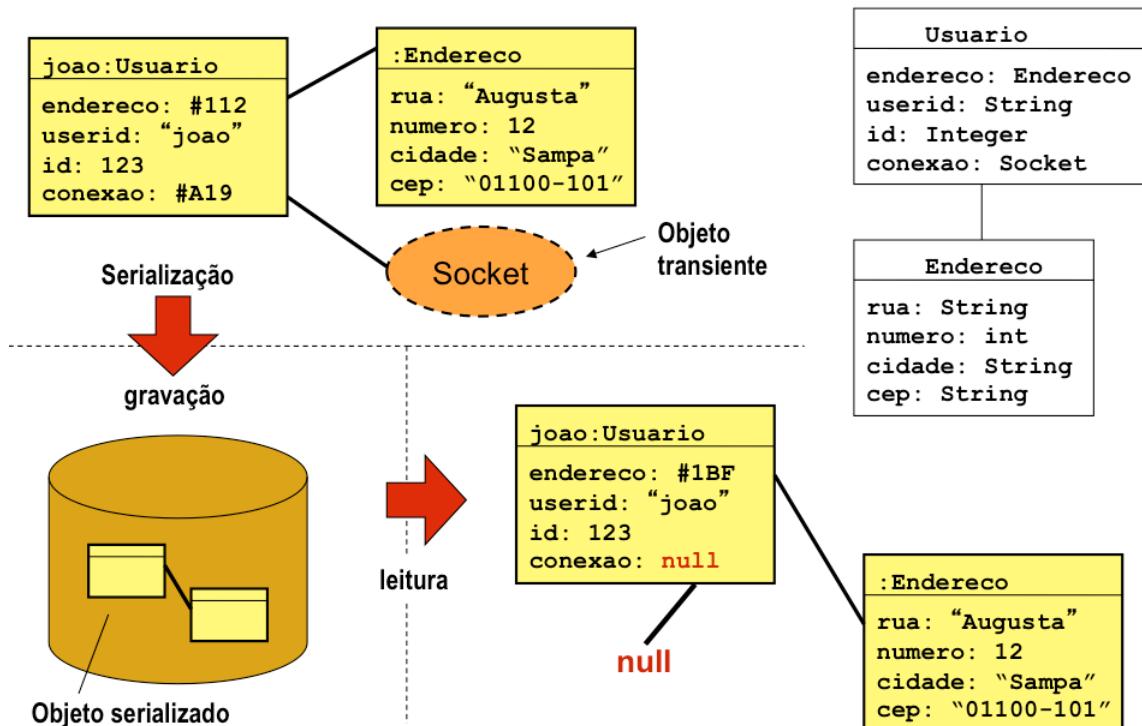
Para extrair dados de um arquivo e recuperar o objeto, usa-se `ObjectInputStream` e `readObject()`:

```
public Jogo abrirJogoExistente(String usuario) {
    File arquivo = new File(usuario + ".jogo");
    ObjectInputStream in =
        new ObjectInputStream(
            new FileInputStream(arquivo));

    Jogo jogo = (Jogo)in.readObject(); // retorna Object
    jogo.setConexao(reabrirConexao(usuario)); // necessário, pois Socket é transient
    return jogo;
}
```

No exemplo acima, o objeto recuperado terá o estado que foi gravado (`usuario="maria"` e `pontuacao=12345`), mas o atributo conexão volta com o valor `null`, por isso é necessário que seu estado seja restaurado por outros meios, como mostrado acima.

O processo de serialização está ilustrado no diagrama abaixo.



## 4 Entrada e Saída Padrão

Java possui o conceito de uma entrada padrão, saída padrão e erro padrão. Esses objetos, que são definidos na classe `System`, são objetos do pacote `java.io`:

<code>System.out</code>	<code>java.io.PrintStream</code>
<code>System.in</code>	<code>java.io.InputStream</code>
<code>System.err</code>	<code>java.io.PrintStream</code>

## 5 Comunicação em rede TCP/IP

O pacote `java.net` oferece várias classes e interfaces que representam componentes de uma rede TCP/IP. Há abstrações para soquetes (`Socket`), URLs e conexões (`URL`, `URLConnection`), endereços Internet (`InetAddress`) e vários outros.

Programar em rede não é simples. Requer não apenas conhecer bem o funcionamento de aplicações TCP/IP como lidar com threads, transações, etc. Em geral é melhor usar Java EE para desenvolver aplicações distribuídas, mas saber como criar soquetes e abrir conexões pode ser útil para desenvolver aplicações simples e testes.

### 5.1 URLs e endereços Internet

Na API Java, a classe `InetAddress` representa um endereço IP. Pode-se obter o endereço usando seu método `getAddress()`. Pode-se tentar obter o nome da máquina através do IP usando o método `getHostName()`, que consultará o serviço de nomes para descobrir o nome correspondente ao endereço.

```
InetAddress end = InetAddress.getByName("info.acme.com");
Byte[] ip   = end.getAddress();
String host = end.getHostName();
```

Existem também classes específicas para lidar com IPv4 e IPv6.

Uma das formas de abrir uma conexão de rede é através de um objeto URL:

```
URL url = new URL("http://www.agonavis.com.br/index.html");

BufferedReader reader =
    new BufferedReader(new InputStreamReader(url.openStream()));
String line = "";
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}
```

Se for necessário obter meta-informação sobre a conexão, URL é insuficiente. A classe `URLConnection` permite que informações de cabeçalho (tipo de dados, comprimento, etc.) sejam recuperadas. `HttpURLConnection` estende `URLConnection` com códigos de protocolos HTTP:

```
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
int resposta = connection.getResponseCode();
if(resposta == HttpURLConnection.HTTP_NOT_FOUND) {
    System.out.println("Página não existe!");
}
```

### 5.2 Soquetes TCP/IP

Soquetes TCP são usados na maioria das aplicações IP que necessitam de garantia de recebimento de pacotes e ordem em que os pacotes são recebidos. Para criar um soquete de dados em Java para enviar dados para um servidor HTTP na máquina `info.acme.com`, pode-se fazer:

```
InetAddress end = InetAddress.getByName("info.acme.com");
Socket con = new Socket(end, 80);
InputStream dados = con.getInputStream();
OutputStream comandos = con.getOutputStream();
// enviar comandos e receber dados do processo remoto...
```

Um Socket disponibiliza um par de streams. Um *InputStream* usado geralmente para recebimento de comandos, e um *OutputStream* usado para envio de dados. Se o protocolo e os dados forem texto, pode-se convertê-los em streams de caracteres:

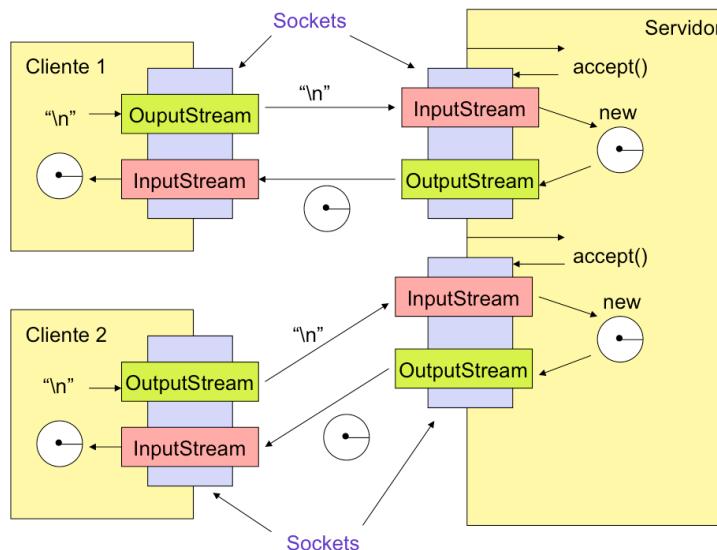
```
Socket con = new Socket("maquina.com.br", 4444);
Reader = new InputStreamReader(con.getInputStream());
Writer = new OutputStreamWriter(con.getOutputStream());
```

Para construir um servidor que fique esperando a conexão de clientes, pode-se usar a classe *ServerSocket*. O soquete de servidor só é necessário enquanto não se obtém um soquete de dados, retornado por seu método *accept()*, que bloqueia a execução do thread até que um soquete de dados seja recebido. Depois de obtida a conexão de dados, o soquete do servidor pode ser descartado ou colocado novamente para escutar outro cliente.

Suponha que na máquina info.acme.com haja um servidor Web escrito em Java. O código do servidor pode ter algo como:

```
ServerSocket escuta = new ServerSocket(80);
while(true) {
    Socket cliente = escuta.accept(); // espera aqui por cliente
    InputStream comandos = cliente.getInputStream();
    OutputStream dados = cliente.getOutputStream();
    // receber commandos, processá-los e enviar dados ao processo remoto...
```

No exemplo acima, o servidor só poderá lidar com um novo cliente depois que o cliente atual tiver terminado seu trabalho. Para que um servidor possa trabalhar com múltiplos clientes ao mesmo tempo, terá que passar os soquetes de dados para novos threads, mantendo o thread principal exclusivamente para escutar e receber novos clientes. A ilustração abaixo mostra um cenário com dois clientes e um servidor usando Sockets e ServerSockets:



## 6 Non-blocking IO (java.nio)

Esta é uma outra API de IO que também faz parte do Java SE. Ela oferece uma alternativa ao modelo orientado a *streams* do *java.io*, com um modelo baseado em *buffers* que não requer o bloqueio de threads para a leitura. Os métodos *read()* bloqueiam o thread esperando que haja dados no stream, que é lido sequencialmente. O modelo *java.nio* trabalha com todos os dados disponíveis em um buffer, e é possível ler dados em diferentes direções. Os recursos são os mesmos, mas para determinadas aplicações, *java.nio* pode ser mais eficiente (ex: aplicações de rede com sockets, com grande demanda de escalabilidade). Para o uso geral, porém, *java.io* é mais simples. Este pacote foi introduzido no Java 1.4 e teve adições importantes no Java 7.

# 10 JDBC

<b>1 O pacote java.sql</b>	<b>2</b>
<b>2 Drivers JDBC</b>	<b>2</b>
<b>3 Como realizar uma conexão</b>	<b>3</b>
3.1 Carga do driver	3
3.2 Conexão usando DriverManager e URL JDBC	3
3.3 Conexão usando DataSource	4
<b>4 Connection, Statement e ResultSet</b>	<b>5</b>
4.1 Usando Statement para criar e destruir tabelas	5
4.2 Usando PreparedStatement para alterar registros	6
4.3 Usando ResultSet para recuperar registros	7
<b>5 Stored procedures</b>	<b>7</b>
<b>6 Fechamento de conexões e tratamento de exceções</b>	<b>8</b>
<b>7 Transações</b>	<b>8</b>
<b>8 Metadados</b>	<b>8</b>

Há várias formas de preservar os dados entre execuções de um programa

- Na memória (em servidores ou sistemas que fazem ativação/passivação)
- Em sistemas de arquivos
- Em sistemas de bancos de dados

Bancos de dados, principalmente bancos de dados relacionais, são uma das formas mais comuns de persistência para dados estruturados que precisam ser rapidamente localizados.

*Structured Query Language* é uma linguagem universal utilizada para acesso a bancos de dados relacionais. É padronizada pela organização ANSI. As versões mais populares são SQL92 e SQL99, que são suportadas por todos os grandes fabricantes de bancos de dados. Há implementações completas e mínimas. As implementações mínimas suportam pelo menos as operações essenciais: CRUD (Create, Retrieve, Update, Delete).

Várias linguagens de propósito geral possuem interfaces para a comunicação com bancos de dados. Existem basicamente dois tipos:

- Interfaces de nível de chamada (call level): ODBC, JDBC onde o SQL é misturado com o código da linguagem (C, Java)

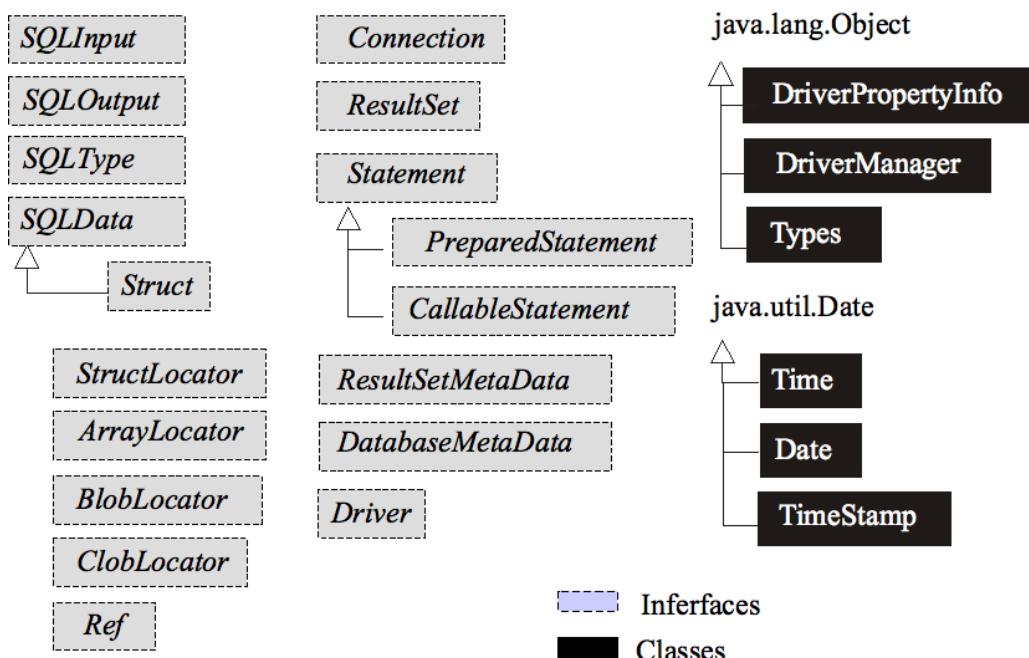
- Interfaces de alto nível, onde o código SQL é ausente: sistemas ORM (Object-Relational Mapping) como Hibernate, TopLink, JPA

Em Java, a forma padrão de acesso a bancos de dados relacionais é através da API JDBC, distribuída no pacote `java.sql`. JDBC é uma interface procedural de nível de chamada, ou seja, as chamadas de SQL são feitas explicitamente dentro do código Java. Isto é diferente da interface JPA, que também é um padrão Java para acesso a bancos de dados, mas é uma interface orientada a objetos, e que gera e esconde o SQL do programador.

O pacote `java.sql` consiste de classes e interfaces para embutir código SQL em métodos. Com JDBC pode-se construir aplicações Java para acesso a qualquer banco de dados SQL.

## 1 O pacote `java.sql`

As classes e interfaces abaixo fazem parte do pacote `java.sql`, que é usado por programadores de aplicações JDBC. Além desse pacote, também existe o pacote `javax.sql` que possui algumas classes e interfaces adicionais.



## 2 Drivers JDBC

Para escrever uma aplicação JDBC é preciso usar o pacote `java.sql` e um driver JDBC que realize a comunicação com um banco específico. Os drivers são implementações da API do JDBC e são distribuídos geralmente como um conjunto de classes empacotadas em um JAR. Um mesmo banco pode ter vários drivers diferentes, para diferentes finalidades. Há também drivers que suportam versões diferentes do SQL ou do JDBC.

Uma aplicação JDBC pode carregar ao mesmo tempo na memória diversos drivers (de fabricantes e versões diferentes). Para determinar qual dos drivers carregados será usado para realizar uma determinada conexão, utiliza-se uma URL que tem o seguinte formato:

```
jdbc:<subprotocol>:<dsn>
```

A aplicação usa o subprotocolo para identificar e selecionar o driver a ser instanciado (geralmente é o nome do fabricante do banco (ex: oracle, mysql, h2, postgres, etc.). O dsn (data source name) é o nome que o subprotocolo utilizará para localizar um determinado servidor ou base de dados. A sintaxe depende do fabricante. Veja alguns exemplos:

```
jdbc:odbc:anuncios
jdbc:oracle:thin:@200.206.192.216:1521:exemplo
jdbc:mysql://alnitak.orion.org/clientes
jdbc:cloudscape:rmi://host:1098/MyDB;create=true
```

## 3 Como realizar uma conexão

### 3.1 Carga do driver

A interface *Driver* é utilizada pelas implementações de drivers JDBC. É preciso ter o driver no Classpath e carregar a classe que implementa essa interface na aplicação que irá utilizá-lo. Numa aplicação Java standalone, isto pode ser feito simplesmente declarando uma variável qualquer com a classe do driver, mas a forma mais indicada é usando o *Classloader* através do método *Class.forName()*, que permite carregar qualquer classe Java a partir do seu nome completo. É preciso verificar com o fabricante para saber qual o nome do Driver que deve ser carregado.

O exemplo abaixo carrega drivers para bancos PostgreSQL, MySQL, H2, HSQLDB e Oracle.

```
Class.forName("org.h2.Driver");
Class.forName("com.mysql.jdbc.Driver");
Class.forName("org.hsqldb.jdbc.JDBCDataSource");
Class.forName("org.postgresql.Driver");
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Para que os drivers sejam carregados, é necessário que o JAR dos quatro bancos estejam configurados no Classpath como dependências de tempo de execução. Veja no apêndice como configurar o pom.xml do projeto Maven para incluir essas dependências automaticamente.

Carregando-se o driver, o próximo passo é obter uma conexão para que se possa realizar a comunicação com o banco de dados.

### 3.2 Conexão usando DriverManager e URL JDBC

A classe *DriverManager* seleciona o banco e a fonte de dados através de três parâmetros: URL JDBC, nome e senha, inicia uma tentativa de conexão e, se houver sucesso, devolve uma referência para essa conexão, que pode ser usada para realizar a comunicação com o banco.

Por exemplo, a linha abaixo retorna ao objeto **con1** uma conexão com uma fonte de dados HSQLDB local chamada **curso**:

```
Connection con1 =
    DriverManager.getConnection ("jdbc:hsqldb:hsqldb://localhost/curso", "sa", "");
```

Já o objeto **con2**, no exemplo abaixo, recebe uma conexão com uma fonte de dados PostgreSQL remota, chamada **vendas**:

```
Connection con2 =
    DriverManager.getConnection ("jdbc:postgresql://200.112.108.9", "admin", "@dm1N");
```

Depois que uma conexão for obtida, e usada, ela deve ser fechada explicitamente, para que o banco possa liberar os recursos alocados. Isto é feito com o método *close()*. Geralmente *close()* é usado dentro de um bloco *finally*, para garantir a execução:

```

try {
    Connection con = ...
    // Usa a conexão
} finally {
    con.close(); // fecha a conexão com o banco
}

```

### 3.3 Conexão usando DataSource

Outra forma de obter a conexão é utilizar a classe `DataSource` e um driver que implemente o pacote `javax.sql`. Os drivers que implementam `javax.sql` fornecem acesso através de um serviço de pool de conexões, que é muito mais eficiente. Esse tipo de acesso é comum em servidores de aplicação, que previamente criam o data source e um número inicial de conexões, e permite que elas sejam usadas, e posteriormente devolvidas ao pool para serem reutilizadas.

Para obter acesso a um `DataSource` é preciso usar um serviço de terceiros. Em servidores de aplicação, `DataSources` são acessíveis através de interfaces padrão, como JNDI, DI ou CDI, mas em programas standalone Java é preciso usar um pacote externo.

Existem vários pacotes populares, distribuídos pelo grupo Apache e outras organizações de código aberto. Alguns exemplos são DBCP, C3PO e Hikari. Este último é considerado atualmente (2015) o mais eficiente deles.

Para usar é preciso incluir o JAR do Hikari como dependência do projeto (veja no apêndice como configurar o `pom.xml` do Maven para carregar o Hikari). Uma vez instalado, é preciso configurar o Hikari para que ele use o banco:

```

HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:hsqldb:hsq://localhost/curso");
config.setUsername("sa");
config.setPassword("");

```

Depois, obtém-se um `DataSource` através de instanciamento simples, passando o objeto `config` como parâmetro:

```
DataSource ds = new HikariDataSource(config);
```

Existem várias configurações que podem ser feitas no pool, para ajustar performance e alocação de recursos.

Uma vez obtido o `DataSource`, ele pode ser usado para obter uma conexão simplesmente usando o método `getConnection()`, que não cria uma conexão nova, mas simplesmente obtém uma conexão existente do pool;

```

try {
    Connection con = ds.getConnection();
    // Usa a conexão
} finally {
    con.close(); // fecha a conexão com o banco
}

```

O método `close()` também deve ser chamado depois que a conexão for usada, porém neste caso ela não será fechada, mas devolvida ao pool para reuso. O pool pré-instancia várias conexões, tornando mais eficiente a alocação e liberação de recursos.

`DataSource` é uma alternativa muito mais eficiente que `DriverManager` e deve ser usada sempre que possível.

## 4 Connection, Statement e ResultSet

Uma vez obtida uma conexão, seja via DriverManager ou DataSource, é possível iniciar a comunicação com o banco. JDBC traz três interfaces fundamentais para realizar essa comunicação. Elas são implementadas em todos os drivers JDBC. Já vimos a primeira.

- **Connection:** Representa uma conexão ao banco de dados, que é retornada pelo DriverManager ou pelo DataSource na forma de um objeto.
- **Statement:** Oferece meios de passar instruções SQL para o bancos de dados. Criada através de uma Connection. Para queries parametrizados geralmente usa-se uma subclasse que é mais eficiente.
- **ResultSet:** É um cursor para os dados recebidos. É retornado como resultado de um query Select.

Operações com essas interfaces potencialmente causam SQLException, que encapsula erros na comunicação com o banco. O tratamento dessa exceção é obrigatório, portanto o código dos exemplos a seguir devem estar obrigatoriamente ou dentro de um bloco try-catch, ou em um método que declara lançar SQLException.

Um Statement é criado a partir de um Connection usando o método createStatement():

```
Statement stmt = con.createStatement();
```

Um Statement possui métodos para executar SQL, como execute(), executeQuery(), executeBatch() e executeUpdate(). Possui também duas sub-interfaces: PreparedStatement e CallableStatement. Para executar uma declaração repetidas vezes, ou para declarações parametrizadas, o ideal é usar PreparedStatement no lugar de Statement, já que esta última permite compilar as instruções SQL tornando assim as consultas mais eficientes, e também são mais adequadas para processamento em lote.

Já o CallableStatement é usado para chamar procedimentos nativos armazenados no banco (stored procedures).

```
PreparedStatement pstmt =
    con.prepareStatement(...);
CallableStatement cstmt = con.prepareCall(...);
```

### 4.1 Usando Statement para criar e destruir tabelas

Um Statement pode ser usado para enviar uma instrução SQL para criar uma tabela no banco. Neste caso, como a instrução será executada uma vez apenas, e não retorna valor, usaremos o método execute(), passando o string SQL como parâmetro:

```
stmt.execute("CREATE TABLE dinossauros "
    + "(codigo INT PRIMARY KEY, "
    + "genero CHAR(20), " + "especie CHAR(20));");
```

Um execute() também é adequado para executar outras operações em tabela, como por exemplo, um DROP:

```
stmt.execute("DROP TABLE dinossauros;");
```

Statement também pode ser usado para fazer inserções e atualizações. Neste caso é comum usar o método executeUpdate() poise le retorna um inteiro informando o número de registros afetados. Por exemplo, a instrução abaixo irá alterar um registro:

```
int linhasModificadas =
    stmt.executeUpdate("INSERT INTO dinossauros "
        + "(codigo, genero, especie) VALUES "
        + "(499,'Fernandosaurus','brasiliensis')");
```

O método `executeUpdate()` também deve ser usado para `DELETE` e `UPDATE`.

Um Statement também permite executar queries. O método `executeQuery()` retorna um objeto `ResultSet`, que é um tipo de Iterador de registros, contendo o resultado de uma consulta:

```
ResultSet cursor =
    stmt.executeQuery("SELECT genero, especie "
        " FROM dinossauros "
        " WHERE codigo = 355");
```

Mas, em uma aplicação típica, inserções, atualizações e queries costumam ser executadas várias vezes, portanto vale a pena reusá-los. O ideal neste caso é um `PreparedStatement`.

## 4.2 Usando PreparedStatement para alterar registros

`PreparedStatement` é um tipo de `Statement`, mas que prepara a declaração SQL antes. Possui os mesmos métodos `execute()`, `executeUpdate()` e `executeQuery()` que `Statement`, mas eles são usados sem argumentos, já que o SQL é passado para um `PreparedStatement` no momento em que ele é criado:

```
PreparedStatement cstmt =
    con.prepareStatement("INSERT INTO Livros (numero,autor,titulo) VALUES(?, ?, ?)");
```

Além disso, um `PreparedStatement` pode declarar e receber parâmetros. Os parâmetros são identificados pelos “?” presentes nas cláusulas da declaração que recebem valores. Os parâmetros podem ser preenchidos por métodos que são chamados depois do instanciamento, e só depois a declaração é executada:

```
cstmt.setInt(1, 18943);
cstmt.setString(2, "Lima Barreto");
cstmt.setString(3, "O Homem que Sabia Javanês");
cstmt.executeUpdate();
```

Esta alternativa é muito mais eficiente quando várias queries similares são enviadas com parâmetros diferentes. Os métodos usados para inserir os valores dos parâmetros incluem como primeiro argumento ou o nome da coluna (string) ou a posição em ordem que inicia em 1 (não em zero). Os nomes são auto-explicativos e têm a forma `setXXX()` onde XXX é um tipo básico do Java mapeado pelo driver a um tipo SQL (por exemplo String, Int, Long, Date, Object etc.).

A tabela abaixo relaciona o mapeamento padrão entre tipos Java e SQL99, e métodos de `PreparedStatement` para setar parâmetros:

Método de PreparedStatement	Tipo Java	Tipo SQL 99
<code>setInt()</code>	<code>Integer/int</code>	<code>INTEGER</code>
<code>setShort()</code>	<code>Short/short</code>	<code>SMALLINT</code>
<code>setByte()</code>	<code>Byte/byte</code>	<code>TINYINT</code>
<code>setLong()</code>	<code>Long/long</code>	<code>BIGINT</code>
<code>setFloat()</code>	<code>Float/floa</code> t	<code>REAL</code>
<code>setDouble()</code>	<code>Double/double</code>	<code>DOUBLE</code>
<code>setBigDecimal()</code>	<code>BigDecimal</code>	<code>NUMERIC</code>
<code>setBoolean()</code>	<code>Boolean/boolean</code>	<code>BIT, BOOLEAN</code>
<code>setString()</code>	<code>String</code>	<code>VARCHAR, LONGVARCHAR</code>

Método de PreparedStatement	Tipo Java	Tipo SQL 99
setDate()	Date	DATE
setTime()	Time	TIME
setTimestamp()	Timestamp	TIME STAMP
setObject()	Object	BLOB, CLOB

Usando esses métodos, o JDBC automaticamente insere o tipo correspondente no query (e evitam a necessidade de fazer concatenação de strings ou lembrar de colocar apóstrofes nos ao usar valores CHAR).

### 4.3 Usando ResultSet para recuperar registros

O método executeQuery(), da interface Statement, retorna um objeto ResultSet, que é um cursor para navegação pelas linhas de uma tabela. ResultSet é um Iterator. Através dele pode-se recuperar as informações armazenadas nas colunas. usando métodos de navegação, que são next(), previous(), absolute(int), first(), last() que retornam boolean.

Para obter todos os registros, uma chamada a next() dentro de um loop é suficiente. Para cada registro obtido, pode-se extrair dados de cada uma de suas colunas, já fazendo a conversão de tipos automaticamente através de vários métodos como getXXX():

- getInt()
- getString()
- getDate()
- etc.

Esses métodos são análogos aos métodos setXXX() de PreparedStatement. O parâmetro pode ser um número (posição da coluna no registro-resultado, iniciando em 1) ou uma String (nome da coluna). O exemplo abaixo ilustra o uso de ResultSet com o método de navegação next() usando os nomes das colunas para extrair dos dados de cada registro.

```
ResultSet rs =
stmt.executeQuery("SELECT Numero, Texto,
+ " Data FROM Anuncios");

while (rs.next()) {
    int x = rs.getInt("Numero");
    String s = rs.getString("Texto");
    java.sql.Date d = rs.getDate("Data");
    // faça algo com os valores obtidos...
}
```

## 5 Stored procedures

Procedimentos desenvolvidos em linguagem proprietária do SGBD (stored procedures) podem ser chamados através de objetos CallableStatement. Os parâmetros são passados da mesma forma que em instruções PreparedStatement. O fragmento abaixo ilustra o uso:

```
con.prepareCall("{ call proc_update(?, ?, ...) }");
con.prepareCall("{ ? = call proc_select(?, ?, ...) }");

CallableStatement cstmt = con.prepareCall("{? = call sp_porAssunto(?)}";
cstmt.setString(2, "520.92");
ResultSet rs = cstmt.executeQuery();
...

```

## 6 Fechamento de conexões e tratamento de exceções

Assim como Connection, Statement, PreparedStatement e ResultSet devem ser fechados após o uso usando o método close():

```
try {
    ResultSet rs = ...
} finally {
    rs.close();
}
```

A exceção SQLException é a principal exceção a ser observada em aplicações JDBC. Use seu método getErrorCode() para obter o código de erro (que é dependente do SGBD) e use getState() para obter estado da conexão:

```
try {
    ...
} catch (SQLException e) {
    LOG.log("Erro no banco: " + e.getErrorCode());
}
```

## 7 Transações

Uma transação é uma unidade indivisível de execução. Uma operação que envolve várias etapas de alteração de registros é um exemplo típico de um procedimento que precisa ser protegido por um contexto transacional. Se todas as etapas executarem com sucesso, o resultado deve ser persistido no banco, mas se alguma etapa falhar, as etapas já realizadas devem ser desfeitas.

As operações do JDBC podem ser configuradas para ocorrerem em um escopo transacional se a conexão for configurada usando:

```
con.setAutoCommit(false);
```

Qualquer Statement, CallableStatement ou PreparedStatement criado e executado a partir da conexão após essa declaração será executada, mas seus resultados não serão mais persistidos imediatamente no banco. Para enviar para o banco um lote de statements executados, é necessário chamar:

```
con.commit();
```

Que só irá realizar as alterações se todas as execuções individuais ocorrerem com sucesso. Caso alguma delas falhar, uma exceção será lançada e o bloco catch deverá chamar

```
con.rollback();
```

Para que as alterações já realizadas sejam desfeitas. (Esta é uma descrição conceitual do funcionamento de uma transação JDBC: o processo exato de como o banco processa os resultados intermediários depende de configurações proprietárias, como por exemplo, os níveis de isolamento).

## 8 Metadados

As classes de metadados permitem obter dados sobre o banco de dados e sobre o resultado de uma consulta. Não são implementações obrigatórias da especificação, o que significa que nem todos os drivers suportam esses recursos. As classes mais importantes são:

Classe **DatabaseMetaData**: permite obter informações relacionadas ao banco de dados

```
Connection con; (...)DatabaseMetaData dbdata = con.getMetaData();
String nomeDoSoftwareDoBanco =
    dbdata.getDatabaseProductName();
```

Classe **ResultSetMetaData**: permite obter informações sobre o ResultSet, como quantas colunas e quantas linhas existem na tabela de resultados, qual o nome das colunas, etc.

```
ResultSet rs; (...)  
ResultSetMetaData meta = rs.getMetaData();  
int colunas = meta.getColumnCount();  
String[] nomesColunas = new String[colunas];  
for (int i = 0; i < colunas; i++) {  
    nomesColunas[i] = meta.getColumnName(i);  
}
```

# Apêndice A - Exercícios e testes

## 1 Exercícios

1. Na listagem abaixo, associe os números dos balões e identifique as partes de um programa em Java preenchendo as lacunas abaixo:

```

import java.util.HashSet;
import java.util.Set;
/**
 * Representa um navio.
 */
public class Navio extends Transporte implements Pesavel {
    private String nome;
    private Set<Carga> cargas = new HashSet<Carga>();
    private double peso;

    public Navio(double peso) {
        this.peso = peso;
    }

    public Navio(String nome, double peso) {
        this.peso = peso;
        this.nome = nome;
    }

    protected void addCarga(Carga carga) {
        cargas.add(carga);
        peso += carga.getPeso();
    }

    public void removeCarga(Carga carga) {
        cargas.remove(carga);
        peso -= carga.getPeso();
    }

    public double getPeso() {
        return peso;
    }

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

```

The diagram shows a Java code snippet with nine numbered callouts (1 through 9) pointing to various parts of the code. Callout 1 points to the first line of the code. Callout 2 points to the word 'Transporte' in the class inheritance. Callout 3 points to the 'implements Pesavel' part. Callout 4 points to the 'cargas' field. Callout 5 points to the constructor with two parameters. Callout 6 points to the 'addCarga' method. Callout 7 points to the 'getPeso' method. Callout 8 points to the 'toString' method. Callout 9 points to the closing brace of the class definition.

( ) Definição de uma **classe**. Nome da classe: \_\_\_\_\_

( ) Declaração de **atributos**.

Tipos dos atributos: \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_

Nomes dos atributos: \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_

- ( ) Anotação  
 ( ) Comentário  
 ( ) Importação de classes  
 ( ) **Parâmetros** de um método ou construtor.  
 Tipos dos parâmetros: 1) \_\_\_\_\_, 2) \_\_\_\_\_
- ( ) Definição de um dos **construtores** da classe  
 ( ) Definição de um dos **métodos** da classe. Nome do método: \_\_\_\_\_  
 ( ) Seqüência de **instruções** dentro de um método
2. Represente os seguintes *conceitos* como classes, desenhando-os em UML e com código em Java
- Um Produto, que contém um código numérico inteiro, um nome e um preço em ponto-flutuante
  - Um Pedido, que contém um código numérico inteiro, um Cliente ao qual ele pertence, uma operação que retorna o total e uma coleção de Itens.
  - Um Item de Pedido, que contém um Produto, uma quantidade e uma operação que retorna o valor total.
  - Uma Pessoa, que possui nome e endereço e operações que permitam mudar o endereço, mas não o nome.
  - Um Usuário, que é uma Pessoa que possui um login e senha.
  - Um Cliente, que é um tipo de Usuário que possui uma lista de Pedidos.
  - Uma Publicação, que é um tipo de produto que possui número de páginas.
  - Um Autor que é um tipo de Pessoa que possui uma ou mais Publicações.
  - Um Livro que é um tipo de Publicação que possui um ou mais Autores.
  - Um Estoque que contém uma coleção de Produtos e operações para adicionar Produtos e retornar os Produtos que estão em estoque.
3. Represente as seguintes classes em UML e Java
- Uma Pessoa tem um nome (String)
  - Uma Porta tem um estado *aberto*, que pode ser *true* ou *false* e pode ser *aberta* ou *fechada* (comportamento)
  - Uma Construcao tem uma finalidade
  - Uma Casa é uma Construcao com finalidade residencial que tem um proprietário Pessoa, um número e um conjunto (vetor) de Portas
4. Crie as seguintes classes
- Um Ponto tem coordenadas x e y inteiras
  - Um Circulo tem um Ponto e um raio inteiro
  - Um Pixel é um tipo de Ponto que possui uma cor

## 2 Revisão de Java Básico

### 2.1 Instruções

- Crie um projeto para cada exercício.
- Use pacotes.
- Siga as convenções da linguagem.
- Escolha um exercício de cada seção.
- Veja os arquivos que existem prontos para cada exercício e utilize-os.

#### 2.1.1 Arquivos fornecidos

Seção I (escolha UM)

- Exercício 1: nada

- Exercício 2: classe LoteriaTeste

#### Seção II (escolha DOIS)

- Exercício 1: classe ContatoTeste
- Exercício 2: arquivo estados.txt
- Exercício 3: arquivo telefones.txt

#### Seção III (escolha UM)

- Exercício 1: classe NaviosTeste
- Exercício 2: classes FormataNomes, FormataNomesSimples, FormataNomesHtml, FormataNomesTeste

#### Seção IV (escolha UM)

- Exercício 1: classe TelefoneBDTeste \*\*\*
- Exercício 2: arquivo telefones.txt

## 2.2 Seção I: básico e sintaxe (escolha UM exercício desta seção)

1. Um ano bissexto é representado por um número inteiro que é divisível por quatro *e* que não é divisível por cem, *ou* que é divisível por quatrocentos. Escreva um programa em Java que liste na tela todos os anos bissextos entre 1900 e 2100, e no final informe quantos anos bissextos há nesse intervalo. O programa deverá também numerar os anos bissextos, como no exemplo abaixo:

```
> java Bissexto
1: 1804
...
24: 1692
25: 1696
Há 25 anos bissextos entre 1800 e 2100.
```

2. O método estático Math.random() retorna um número aleatório a cada chamada. O número é um double entre zero e um. Por exemplo: para retornar um número entre 1 e 100, multiplique o valor retornado por uma chamada a Math.random() por 100, passe para o método Math.ceil() que arredonda para o próximo inteiro, e faça o cast com (int). Utilizando essa informação, escreva um programa que gere dez séries de seis números entre 1 e 50 e imprima na tela, da seguinte forma:

```
Aposta 1: [28][40][9][23][48][5]
Aposta 2: [42][21][31][25][46][13]
Aposta 3: [6][17][3][31][36][1]
Aposta 4: [27][1][22][10][26][14]
...
```

*Dicas:*

- 1) Aproveite a classe **LoteriaTeste** fornecida. Leia o código e entenda o código antes de escrever qualquer coisa.
- 2) Crie uma classe **Loteria** contendo um método **sorteio()** que retorne um array de inteiros. Cada chamada a sorteio deve devolver um array de seis posições preenchido com números aleatórios entre 1 e 50. O método sorteio cria o array de seis posições, e chama Math.random() (ou Random.nextInt()) seis vezes para preencher os elementos do array. Não é preciso verificar (nesta versão) se números foram repetidos na aposta. O programa principal (LoteriaTeste) instancia essa classe e chama sorteio() dez vezes para imprimir os resultados na tela.

3. Escreva um programa em Java que leia o arquivo **nomes.txt**, que contém uma lista de nomes, um por linha. Preencha uma lista ou um array com esses nomes, liste todos na tela e depois sorteie um deles aleatoriamente e imprima na tela. A saída deve ser algo semelhante à listagem abaixo:

```
> java Sorteio
** Participantes **
Albert Einstein
Marie Curie
Richard Feynman
Carl Sagan

** Vencedor **
Carl Sagan
```

## 2.3 Seção II: classes e objetos (escolha DOIS exercícios desta seção)

1. Implemente a hierarquia de classes abaixo

Uma **UnidadeDaFederacao** é formada por

- Um nome, que é String
- Uma abreviação, que é String

Um **Endereco** é formado por

- Um logradouro, que é String
- Um número, que é int
- Um complemento, que é String
- Uma cidade, que é String
- Um estado que é UnidadeDaFederacao

Um **Telefone** é formado por:

- Código de país (pais), que é um int
- Código de área (ddd), que é um int
- Número (numero), que é um int.

Um **Contato** é formado por

- Um nome, que é String
- Um email, que é String
- Um telefoneResidencial, que é Telefone
- Um telefoneComercial, que é Telefone
- Um celular, que é Telefone
- Um endereço, que é Endereco.

Implemente a hierarquia de classes, com `toString()`, `equals()` e `hashCode()`, para que o programa **ContatoTeste**, que cria cinco contatos e os lista na tela, funcione sem erros e imprima, para cada contato, uma saída semelhante a esta:

Zé Colmeia (ze@colmeia.org) tel: +55(11)95783432, Av. Angelica, 123, Sala 321, São Paulo, SP

*Dica:* implemente os `toString()` de cada objeto e chame-os dentro do `toString()` de Contato.

2. Uma **UnidadeDaFederacao** possui um **nome** e uma **abreviacao**. Ambos são Strings. Para criar uma UnidadeDaFederacao é preciso informar o nome da unidade, e sua abreviação, por exemplo:

```
Estado sp = new Estado("São Paulo", "SP");
```

Crie uma classe `UnidadeDaFederacao` com métodos `getAbreviacao()` e `getNome()` retornando esses campos.

- Implemente `toString()` para retornar a abreviação, espaço, nome.
- Implemente `equals()` e `toString()`
- Implemente `Comparable<T>` e `compareTo()` ordenando os estados pela abreviação.
- Crie um comparador `EstadoComparador` que permita ordenar os estados pelo nome.
- Escreva uma classe **UnidadeDaFederacaoTeste** que crie um array ou lista de estados e imprima-os na tela em ordem natural (pela abreviação) e pela ordem alfabética dos nomes. O resultado deve ser similar ao mostrado abaixo (use arquivo `estados.txt`):

Estados (ordenado pelas abreviações):

```
AC Acre
AM Amazonas
AP Amapá
...
TO Tocantins
```

Estados (em ordem alfabética):

```
AC Acre
AP Amapá
AM Amazonas
...
TO Tocantins
```

3. Um **Telefone** é formado por um código de país (`pais`), que é um int, um código de área (`ddd`), que é um int e um número (`numero`), que é um int. Para criar um Telefone é preciso informar o seu número:

```
Telefone casa = new Telefone(98569834);
```

Mas deve ser possível também informar com DDD:

```
Telefone casa = new Telefone(11, 98569834);
```

Ou com código de país:

```
Telefone casa = new Telefone(55, 11, 98569834);
```

Se for passado apenas o número, ou apenas número e DDD, deve-se considerar os valores default: 11, para ddd, e 55, para país.

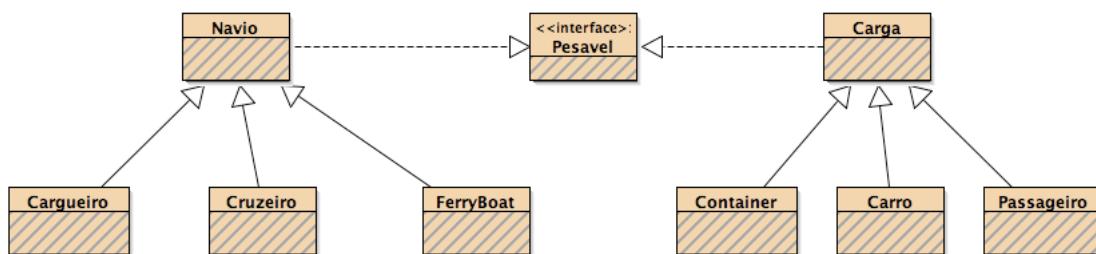
- Crie uma classe `Telefone` com métodos `int getTelefone()` que retorna o número inteiro, `getDdd()` que retorna apenas o DDD, `getPais()`, que retorna o código de país.
- Implemente `toString()` para retornar o telefone formatado da forma `+pais(ddd)numero`.
- Implemente `equals()` e `toString()`
- Implemente `Comparable` e `compareTo()` ordenando os telefones pelo número.
- Crie um comparador `DDDComparador` que permita ordenar os telefones pelo ddd.
- Escreva uma classe **TelefoneTeste** que crie um array ou lista de 10 telefones e imprima-os na tela em ordem natural (pelo número) e pela ordem dos códigos de DDD. O resultado deve ser similar ao mostrado abaixo:

Telefones (ordenado pelo número):  
 +1(477)8859483  
 +55(11)85934358  
 +39(6)89340523

Telefones (ordenado pela área ddd):  
 +39(6)89340523  
 +55(11)85934358  
 +1(477)8859483

## 2.4 Seção III: herança, interfaces, polimorfismo (escolha UM exercício)

1. Implemente a seguinte hierarquia de classes em Java.



Crie os métodos e construtores corretos para que a classe **NaviosTeste** rode sem erros e obtenha a seguinte saída:

```

Peso total dos navios (ton): 280.0
Peso total dos navios (ton): 285.0
Peso total dos navios (ton): 335.0
Peso total dos navios (ton): 305.0
  
```

2. A classe **FormataNomes** é abstrata. Ela recebe um array de Strings e transforma em um String formatado para exibição, incluindo texto antes e depois da lista, e antes e depois de cada item da lista.

A classe **FormataNomesSimples** é uma implementação básica da classe FormataNomes, que implementa três dos quatro métodos abstratos retornando Strings vazios, e o método fimDaLinha() retornando "\n".

```

Albert Einstein
Marie Curie
Richard Feynman
Carl Sagan
  
```

A classe **FormataNomesHtml** é uma implementação da classe FormataNomes que implementa seus métodos de forma a gerar uma tabela em HTML:

```


|                 |
|-----------------|
| Albert Einstein |
| Marie Curie     |
| Richard Feynman |
| Carl Sagan      |


```

- Modifique o código de FormataNomesTeste, que está usando uma implementação de FormataNomesSimples, para que utilize FormataNomesHtml
- Escreva uma classe **FormataNomesConsole** que gere a seguinte String de saída:

```

** Início da lista de nomes **
- Albert Einstein
- Marie Curie
- Richard Feynman
- Carl Sagan
** Fim da lista de nomes **
  
```

Para rodar, use a classe **FormataNomesTeste**.

## 2.5 Parte V: JDBC (escolha UM exercício desta seção)

Preparação (para ambos os exercícios):

A. Crie uma base de dados contendo uma tabela com a seguinte estrutura

```
create table telefone (
    pais integer,
    ddd integer,
    numero integer
);
```

B. Mapeie a base a uma fonte ODBC com o nome **telefone**. A sua URL JDBC deve ser **jdbc:odbc:telefone**. Teste a conexão com o banco.

*Observação:* a tabela pode ser criada tanto em Java como usando as ferramentas do seu banco de dados.

1. DAO com operações básicas

- a. Crie uma exceção chamada **TelefoneException**.
- b. Crie uma interface **TelefoneDAO** com as seguintes operações:

```
void criarTelefone(Telefone tel) throws TelefoneException;
void removerTelefone(Telefone tel) throws TelefoneException;
Collection<Telefone> listarTelefones() throws TelefoneException;
```

c. Crie uma implementação de **TelefoneDAO** chamada **JDBCTelefoneDAO** que implemente os métodos acima.

d. Rode a classe **TelefoneBDTeste**.

2. Classe Java para testar o banco.

- a. Escreva uma classe **CriarTelefones** que quando executada, inclua 5 novos telefones no banco de dados.
- b. Escreva uma classe **ListarTelefones** que, quando executada, grave os dados de cada registro em um objeto **Telefone** e imprima na tela a lista de telefones.
- c. Escreva uma classe **ImportarTelefones** que, quando executada, leia o arquivo **telefones.txt** que contém um telefone por linha no formato *+pais(ddd)numero*. Extraia o código do país, ddd e número de cada linha e crie um objeto **Telefone** para cada um. Depois, inclua cada objeto **Telefone** no banco.

*Dica:* use métodos da classe **String** como **substring()**, **indexOf()**, **charAt()** ou **split()** para tratar o string. Faça testes antes de gravar no banco para saber se a separação dos campos foi correta.

## 3 Testes

(271) Considere as seguintes classes:

```
public class Jogada {
    public int contagem;
    public void jogar(Jogada umaJogada) {
        umaJogada.contagem++;
    }
}
public class TesteJogada {
    public static void main(String args[]) {
        Jogada j = new Jogada();
        j.contagem = 100;
```

```

        j.jogar(j);
        System.out.println(j.contagem);
    }
}

```

Que valor será impresso quando a classe `TesteJogada` for executada?

- a) 0
- b) 100
- c) 101
- d) 102
- e) 1

(272) Considere as seguintes classes:

```

public class Jogada {
    public int contagem;
    public void jogar(Jogada umaJogada) {
        umaJogada.contagem++;
    }
}
public class TesteJogada {
    public static void main(String args[]) {
        Jogada j = new Jogada();
        j.contagem = 100;
        j.jogar( new Jogada() );
        System.out.println(j.contagem);
    }
}

```

Que valor será impresso quando a classe `TesteJogada` for executada?

- a) 0
- b) 100
- c) 101
- d) 102
- e) 1

(273) Considere as seguintes classes:

```

public class Jogada {
    public int contagem;
    public void jogar(Jogada umaJogada) {
        contagem++;
        umaJogada.contagem++;
    }
}
public class TesteJogada {
    public static void main(String args[]) {
        Jogada j = new Jogada();
        j.contagem = 100;
        j.jogar(j);
        System.out.println(j.contagem);
    }
}

```

Que valor será impresso quando a classe `TesteJogada` for executada?

- a) 0
- b) 100
- c) 101
- d) 102
- e) 1

(274) Considere as seguintes classes:

```

public class Jogada {
    public int contagem;
    public void jogar(int contagem) {
        this.contagem += contagem ;
    }
}

public class TesteJogada {
    public static void main(String args[]) {
        int contagem = 100;
        Jogada j = new Jogada();
        j.jogar(contagem);
        System.out.println(contagem);
    }
}

```

Que valor será impresso quando a classe TesteJogada for executada?

- a) 0
- b) 100
- c) 101
- d) 102
- e) 1

(275) Considere as seguintes classes:

```

public class Jogada {
    public int contagem;
    public void jogar(int contagem) {
        contagem += 1 ;
    }
}

public class TesteJogada {
    public static void main(String args[]) {
        int contagem = 100;
        Jogada j = new Jogada();
        j.jogar(contagem);
        System.out.println(j.contagem);
    }
}

```

Que valor será impresso quando a classe TesteJogada for executada?

- a) 0
- b) 100
- c) 101
- d) 102
- e) 1

(276) Considere as seguintes classes:

```

public class Jogada {
    public int contagem;
    public void jogar(int contagem) {
        this.contagem += contagem ;
    }
}

public class TesteJogada {
    public static void main(String args[]) {
        int contagem = 100;
        Jogada j = new Jogada();
        j.jogar(contagem);
        System.out.println(j.contagem);
    }
}

```

```

    }
}

```

Que valor será impresso quando a classe TesteJogada for executada?

- a) 0
- b) 100
- c) 101
- d) 102
- e) 200

(277) Qual o valor final de x no programa abaixo?

```

class StaticTest {
    private static int x = 100;
    private int y = 101;

    public static void main(String [] args) {
        StaticTest st1 = new StaticTest();
        st1.x++;
        st1.y++;
        StaticTest st2 = new StaticTest();
        st2.x++;
        st2.y++;
        StaticTest.x++;
        System.out.println("x = " + x);
    }
}

```

- a) 100
- b) 101
- c) 102
- d) 103
- e) Não vai compilar porque a última linha causa um erro

(141) Considere a seguinte hierarquia de classes e fragmentos de código:

```

java.lang.Throwable
    /
    java.lang.Error      \           java.lang.Exception
    /                   \
java.lang.OutOfMemoryError      java.io.IOException
                                /           \
                                \           \
                                java.io.FileNotFoundException StreamCorruptedException

```

```

1. try {
2.   FileInputStream f = new FileInputStream("circulo.shp");
3.   Object o = in.readObject(); // in é um ObjectInputStream legal
4.   System.out.println("Arquivo lido");
5. }
6. catch (StreamCorruptedException e) {
7.   System.out.println("Arquivo corrompido");
8. }
9. catch (IOException e) {
10.   System.out.println("Erro na leitura");
11. }
12. catch (Exception e) {
13.   System.out.println("Ocorreu uma exceção");
14. }
15. finally {
16.   System.out.println("Concluindo");
17. }
18. System.out.println("Seguindo em frente");

```

Quais linhas (marque uma, nenhuma ou mais de uma) abaixo serão impressas se a linha 2 no código acima provocar um FileNotFoundException?

- a) Arquivo lido
- b) Erro na leitura
- c) Arquivo corrompido
- d) Ocorreu uma exceção
- e) Concluindo
- f) Seguindo em frente

(143) Considere a seguinte hierarquia de classes e fragmentos de código:

```

        java.lang.Throwable
                    /           \
      java.lang.Error       java.lang.Exception
                    /           \
java.lang.OutOfMemoryError     java.io.IOException
                                /           \
                java.io.FileNotFoundException StreamCorruptedException

```

```

1. try {
2.   FileInputStream f = new FileInputStream("circulo.shp");
3.   Object o = in.readObject(); // in é um ObjectInputStream legal
4.   System.out.println("Arquivo lido");
5. }
6. catch (StreamCorruptedException e) {
7.   System.out.println("Arquivo corrompido");
8. }
9. catch (IOException e) {
10.   System.out.println("Erro na leitura");
11. }
12. catch (Exception e) {
13.   System.out.println("Ocorreu uma exceção");
14. }
15. finally {
16.   System.out.println("Concluindo");
17. }
18. System.out.println("Seguindo em frente");

```

*Quais linhas (marque uma, nenhuma ou mais de uma) abaixo serão impressas se as linhas 2 e 3 executarem sem provocar exceção?*

- a) Arquivo lido
- b) Erro na leitura
- c) Arquivo corrompido
- d) Ocorreu uma exceção
- e) Concluindo
- f) Seguindo em frente

(144) Considere a seguinte hierarquia de classes e fragmentos de código:

```

        java.lang.Throwable
                    /           \
      java.lang.Error       java.lang.Exception
                    /           \
java.lang.OutOfMemoryError     java.io.IOException
                                /           \
                java.io.FileNotFoundException StreamCorruptedException

```

```

1. try {
2.   FileInputStream f = new FileInputStream("circulo.shp");
3.   Object o = in.readObject(); // in é um ObjectInputStream legal
4.   System.out.println("Arquivo lido");
5. }
6. catch (StreamCorruptedException e) {
7.   System.out.println("Arquivo corrompido");
8. }
9. catch (IOException e) {

```

```

10. System.out.println("Erro na leitura");
11. }
12. catch (Exception e) {
13.     System.out.println("Ocorreu uma exceção");
14. }
15. finally {
16.     System.out.println("Concluindo");
17. }
18. System.out.println("Seguindo em frente");

```

Quais linhas (marque uma, nenhuma ou mais de uma) abaixo serão impressas se a linha 2 no código acima provocar um OutOfMemoryError?

- a) Arquivo lido
- b) Erro na leitura
- c) Arquivo corrompido
- d) Ocorreu uma exceção
- e) Concluindo
- f) Seguindo em frente

(253) Considere as seguintes classes:

```

public class Registro {
    public int numero;
    public static int contador;
    public Registro() {
        numero = ++contador;
    }
}

public class TesteRegistro {
    public static void main(String[] args) {
        int resultado = /* instrucao */;
        System.out.println(resultado);
    }
}

```

Qual das opções de código abaixo, ao ser colocado no lugar do comentário /\* instrução \*/ , fará com que a execução de TesteRegistro imprima o número do último registro criado?

- a) (new Registro()).numero
- b) numero
- c) contador
- d) Registro.numero
- e) super.numero

(252) Considere as seguinte classes:

```

public class Registro {
    public int numero;
    public static int contador;
    public Registro() {
        numero = ++contador;
    }
}

public class TesteRegistro {
    public static void main(String[] args) {
        int resultado = /* instrucao */;
        System.out.println(resultado);
    }
}

```

O que deve ser colocado no lugar do comentário /\* instrução \*/ para que a execução de TesteRegistro imprima o número de registros criados até o momento sem que isto cause a criação de novos objetos?

- a) (new Registro()).numero
- b) numero
- c) contador
- d) Registro.contador
- e) Registro.numero

(251) Considere a seguinte classe:

```
public class Registro {}
```

Marque a alternativa falsa:

- a) A classe Registro não tem construtor
- b) A classe Registro tem construtor e seu construtor contém uma instrução que chama o construtor da classe java.lang.Object
- c) A classe Registro estende java.lang.Object
- d) A classe Registro tem um método chamado public String toString() que retorna um texto descritivo sobre o objeto
- e) A classe Registro compila e pode ser usada por outras classes para declarar variáveis do tipo Registro

(211) Qual das classes abaixo *não* implementa o padrão de design *iterator* (não estamos falando da interface, mas no padrão de design, na funcionalidade de um iterator)?

- a) java.util.Enumeration
- b) java.util.Iterator
- c) java.sql.ResultSet
- d) java.util.StringTokenizer
- e) java.util.ArrayList

(201) A instrução

```
File arquivo = new File("arquivo.txt");
```

- a) Cria um novo arquivo de texto vazio chamado arquivo.txt no sistema de arquivos local
- b) Cria um novo diretório chamado arquivo.txt no sistema de arquivos local
- c) Cria um objeto com referência chamada arquivo mas não cria nenhum arquivo no sistema de arquivos local
- d) Cria um novo arquivo chamado arquivo.txt no sistema de arquivos local que poderá ou não vir a conter texto, dependendo do tipo do stream usado para preenchê-lo
- e) Cria um novo arquivo no sistema de arquivos local que será um arquivo de texto se o sistema operacional reconhecer a extensão ".txt".

(191) Que classe pode ser usada para carregar uma imagem JPEG para dentro de uma aplicação Java?

- a) InputStream
- b) Reader
- c) Writer
- d) OutputStream
- e) System

(192) Qual das classes abaixo é a mais indicada para carregar um arquivo de texto para dentro de uma aplicação Java?

- a) InputStream
- b) Reader
- c) Writer

- d) OutputStream
- e) System

(193) Qual das classes abaixo é a mais indicada para gravar um texto em um arquivo a partir de uma aplicação Java?

- a) InputStream
- b) Reader
- c) Writer
- d) OutputStream
- e) System

(194) Qual das classes abaixo é a mais indicada para gravar uma imagem gerada por uma aplicação Java em um arquivo?

- a) InputStream
- b) Reader
- c) Writer
- d) OutputStream
- e) System

(181) Considere o arquivo *cidades.txt*, localizado em caminho acessível por uma aplicação Java, com o seguinte conteúdo:

```
Rio de Janeiro
Bruxelas
São Petersburgo
```

O que será gravado na variável *texto* depois de executado o seguinte código?

```
BufferedReader reader =
    new BufferedReader(new FileReader("cidades.txt"));
String texto = "Texto lido: " + reader.readLine();
reader.close();
```

- a) Rio de Janeiro
- b) São Petersburgo
- c) R
- d) o
- e) O código provoca erro de compilação pois não é possível converter char em String.

(182) Considere o arquivo *cidades.txt* com o seguinte conteúdo:

```
Rio de Janeiro
Bruxelas
São Petersburgo
```

O que será gravado na variável *texto* depois de executado o seguinte código?

```
Reader reader = new FileReader("cidades.txt");
String texto = "Texto lido: " + reader.read();
reader.close();
```

- a) Rio de Janeiro
- b) São Petersburgo
- c) R
- d) o
- e) O código provoca erro de compilação pois não é possível converter char em String.

(171) Qual método de `java.sql.Statement` deve ser usado para enviar um `SELECT` para um banco de dados e obter sua resposta?

- a) execute()
- b) executeUpdate()

- c) executeQuery()
- d) Qualquer um deles
- e) Comandos são enviados via `java.sql.Connection` e não `Statement`

(172) Qual método de `java.sql.Statement` deve ser usado para enviar um `INSERT` para um banco de dados e obter como resposta o número de linhas incluídas?

- a) execute()
- b) executeUpdate()
- c) executeQuery()
- d) Qualquer um deles
- e) Comandos são enviados via `java.sql.Connection` e não `Statement`

(132) Considere as seguintes classes

```
public class Saudacoes {
    public String obrigado() { return "Obrigado!"; } //...
}
public class SaudacaoRussa extends Saudacoes {
    public String obrigado() { return "Спасибо!"; } //...
}
public class SaudacaoItaliana extends Saudacoes {
    public String obrigado() { return "Grazie!"; } //...
}
public class SaudacaoAlema extends Saudacoes {
    public String obrigado() { return "Danke!"; } //...
}

public class Carta {
    public static void escrever(Saudacoes s, /*...*/) {
        //...
        System.out.println(s.obrigado());
    }
}
```

e o trecho de código abaixo:

```
SaudacaoRussa spacibo = new SaudacaoAlema();
Carta.escrever(spacibo);
```

O que será impresso (escolha uma alternativa)?

- a) Obrigado!
- b) Danke!
- c) Grazie!
- d) Спасибо!
- e) Haverá um erro de compilação devido à incompatibilidade entre os tipos `SaudacaoRussa` e `SaudacaoAlema`.

(131) Considere as seguintes classes

```
public class Saudacoes {
    public String obrigado() { return "Obrigado!"; } //...
}
public class SaudacaoRussa extends Saudacoes {
    public String obrigado() { return "Спасибо!"; } //...
}
public class SaudacaoItaliana extends Saudacoes {
    public String obrigado() { return "Grazie!"; } //...
}
public class SaudacaoAlema extends Saudacoes {
    public String obrigado() { return "Danke!"; } //...
}

public class Carta {
    public static void escrever(Saudacoes s, /*...*/) {
```

```

    //...
    System.out.println(s. obrigado());
}
}

```

e o trecho de código abaixo:

```

SaudacaoAlema grazie = new SaudacaoAlema();
Carta.escrever(grazie);

```

O que será impresso (escolha uma alternativa)?

- a) Obrigado!
- b) Danke!
- c) Grazie!
- d) Спасибо!
- e) Haverá um erro de compilação devido à incompatibilidade entre os tipos `Saudacoes` e `SaudacaoAlema`.

(121) Considere o trecho de código abaixo onde `s1` e `s2` são Strings:

```
boolean b = (s1 == s2);
```

Marque a alternativa correta:

- a) O código provoca erro de compilação
- b) `b` será `true` se os Strings forem iguais, ou seja, se tiverem os mesmos caracteres na mesma ordem
- c) `b` será `true` se os Strings forem o mesmo objeto, ou seja, se as referências armazenadas em `s1` e `s2` corresponderem ao mesmo número
- d) `b` será `true` se String tiver implementado o método `equals()`
- e) `b` nunca será `true`

(111) Qual das conversões de tipos abaixo é válida?

- a) `int z = 10 + 12.3;`
- b) `double d = 9 * 'a' / 2;`
- c) `boolean true = 1;`
- d) `long m = (double) 12;`
- e) `byte b = 100 + 200;`

(112) Qual das conversões de tipos abaixo é válida?

- a) `int z = 10 + 12.3;`
- b) `char c = 9 * 'a' / 2;`
- c) `boolean true = 1;`
- d) `long m = (double) 12;`
- e) `byte b = (byte) (100 + 200);`

(113) Qual das conversões de tipos abaixo é válida?

- a) `float f = 10 + 12.3f;`
- b) `char c = 9 * 'a' / 2;`
- c) `boolean true = 1;`
- d) `long m = (double) 12;`
- e) `byte b = 100 + 200;`

(101) O que acontece se a palavra-chave `final` estiver presente como um dos modificadores na declaração de um método?

- a) O método não pode mais ser usado por subclasses
- b) O método não é mais visível fora do seu pacote
- c) O método não pode ser sobreposto em subclasses
- d) O método precisa ser sobreposto para que possa ser usado

e) A classe precisa ser declarada como `final`

(102) O que acontece se na declaração de um método não houver um modificador `private`, `public` ou `protected`?

- a) O método não pode mais ser usado por subclasses
- b) O método não é mais visível fora do seu pacote
- c) O método é visível em qualquer lugar, dentro ou fora do pacote
- d) O método precisa ser sobreposto para que possa ser usado
- e) Haverá um erro de compilação pois um modificador de acesso é obrigatório

(103) O que acontece se a palavra-chave `abstract` estiver presente como um dos modificadores na declaração de um método?

- a) O método não pode mais ser usado por subclasses
- b) O método só pode ser usado em suas subclasses
- c) O método deve ter um corpo (entre chaves) vazio
- d) O método precisa ser sobreposto na primeira subclasse
- e) A classe precisa ser declarada como `abstract`

(91) Que palavra-chave é usada para transformar um atributo em constante?

- a) `const`
- b) `private`
- c) `static`
- d) `final`
- e) `abstract`

## Apêndice B – Referências

---

1. Ken Arnold, James Gosling, David Holmes, *The Java Programming Language, 4th Edition*. Addison-Wesley, 2005. *Clássico livro com cobertura abrangente da linguagem Java até a data da última atualização.*
2. Herbert Schildt. *Java: The Complete Reference, Ninth Edition*. Oracle Press, Mc-Graw-Hill Education, 2014. *Provavelmente o livro mais abrangente publicado atualmente sobre a linguagem Java.*
3. Raymond Gallardo, Scott Hommel, Sowmya Kannan, Joni Gordon, Sharon Zakhour. *The Java Tutorial – A Short Course on the Basics. Sixth Edition*. Oracle Java Series. Addison-Wesley, 2014. Disponível online no site da Oracle em [docs.oracle.com/javase/tutorial/](http://docs.oracle.com/javase/tutorial/)
4. Bill Joy, Guy Stelle Jr, Gilad Bracha, James Gosling, Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Oracle Java Series. Addison-Wesley, 2014. Disponível online no site da Oracle em [docs.oracle.com/javase/specs/](http://docs.oracle.com/javase/specs/)
5. Benjamin J. Evans, David Flanagan. *Java In a Nutshell, 6th Edition*. O'Reilly and Associates, 2014. *Referência clássica e abrangente sobre a linguagem.*
6. Joshua Bloch, *Effective Java, 2nd Edition*. Addison-Wesley, 2008. *Explora padrões, boas práticas e anti-patterns sobre Java básico.*