



Centro de Capacitação em TI

Linguagem de
programação

JAVA

Centro de Capacitação em TI



Centro de Capacitação em TI

Conteúdo

Introdução à linguagem java.....	6
A história de java	6
Características da linguagem	7
Plataformas	7
Instalando o JDK.....	8
Instalando o eclipse	14
Imprimindo HelloWorld	16
Princípios básicos.....	21
Membros de classe	21
Método main	21
Declaração e controle de acesso	23
Regras para identificadores.....	23
Convenções de código	24
Padrões JavaBeans	24
Declaração de classes.....	25
Declaração de interfaces	25
Declaração de membros de classe.....	26
Declaração de construtores de classe.....	27
Variáveis	27
Intervalo e tamanhos de primitivos.....	28
Declaração de arrays	28
Declaração de enums	28
Orientação a objetos.....	29
Classe	29
Objeto.....	29
Método	29
Atributo	30
Pacotes	30
Encapsulamento	30
Herança	31
Polimorfismo.....	32
Sobrecarga/Sobrescrita	32
Conversão de variáveis de referência	33
Interfaces.....	33
Associação	34
Abstração	34

Atribuições	35
Literais, atribuições e variáveis.....	35
Arrays.....	37
Wrappers	38
Garbage collector	42
Operadores	44
Atribuição	44
Comparação	45
Igualdade.....	45
Aritméticos.....	47
Lógicos.....	48
Controle de fluxo, exceções e assertivas.....	49
if e switch.....	49
Loops e iteradores	51
Exceções	54
Assertivas.....	57
String, E/S, formatação e parsing	58
String, StringBuilder e StringBuffer.....	58
Manipulação de arquivos	60
Serialização	65
Data, número e moeda	70
Parsing, Tokenização e formatação	73
Genéricos e conjuntos	78
hashCode() e equals()	78
Conjuntos	81
Tipos genéricos	89
Classes internas	91
Comum.....	92
Locais de método	93
Anônimas	93
Estáticas aninhadas	94
Threads.....	95
Iniciando threads	96
Estados e transições	96
Sincronização	98
Interação entre threads	99
Desenvolvimento	100

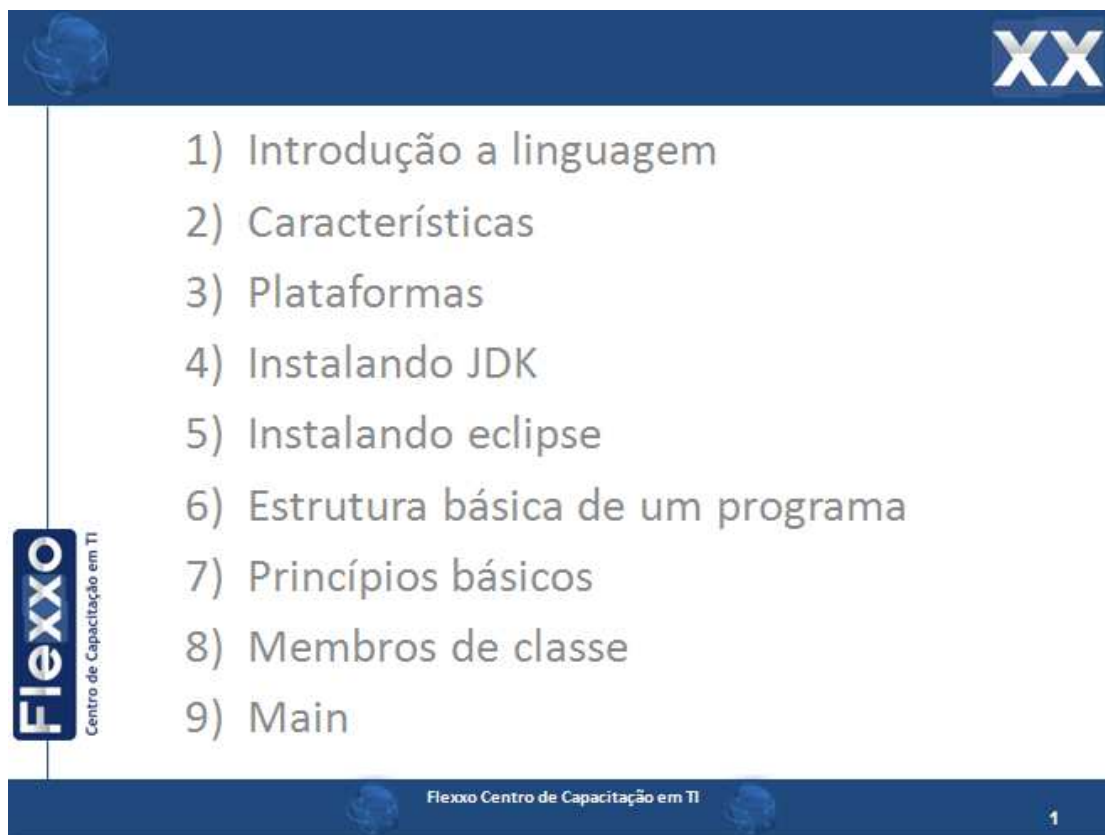
Comandos javac e java.....	100
Arquivos jar	103
Imports estáticos	103



Centro de Capacitação em TI

Introdução à linguagem java

Neste capítulo veremos:



- 1) Introdução a linguagem
- 2) Características
- 3) Plataformas
- 4) Instalando JDK
- 5) Instalando eclipse
- 6) Estrutura básica de um programa
- 7) Princípios básicos
- 8) Membros de classe
- 9) Main

A história de java

Reconhecendo o crescimento dos microcomputadores e da internet, em meados de 1991 a Sun Microsystems financiou uma pesquisa corporativa com codinome GREEN. O projeto resultou no desenvolvimento de uma linguagem baseada em C e C++ que seu criador, James Gosling, chamou de OAK (Carvalho) em homenagem a uma árvore que dava para janela do seu escritório na Sun. Descobriu-se mais tarde que já havia uma linguagem de computador chamada OAK. Quando uma equipe da Sun visitou uma cafeteria local, o nome JAVA (cidade de origem de um tipo de café importado) foi sugerido e pegou.

Mas o projeto GREEN atravessava algumas dificuldades. O mercado para dispositivos eletrônicos inteligentes destinados ao consumidor final não estava se desenvolvendo tão rapidamente como a Sun tinha previsto. Pior ainda, um contrato importante pelo qual a Sun competia fora concedido a outra empresa. Então o projeto estava em risco de cancelamento. Por pura sorte, a World Wide Web explodiu em popularidade em 1993 e as pessoas da Sun viram o imediato potencial de utilizar java para criar páginas na WEB, com o chamado conteúdo dinâmico. Isso deu nova vida ao projeto. Em maio de 1995 a Sun anunciou JAVA formalmente em uma conferência importante. Normalmente, um evento como este não teria gerado muita atenção. Entretanto, JAVA gerou interesse imediato na comunidade comercial por causa do

fenomenal interesse pela internet. JAVA é agora utilizado para criar páginas na WEB com conteúdo interativo e dinâmico, para desenvolver aplicativos corporativos de larga escala, para aprimorar a funcionalidade de servidores da internet, fornecer aplicativos para dispositivos destinados ao consumidor final. Mais tarde em 2009 ao Oracle acabou comprando a SUN.

Características da linguagem

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

Plataformas

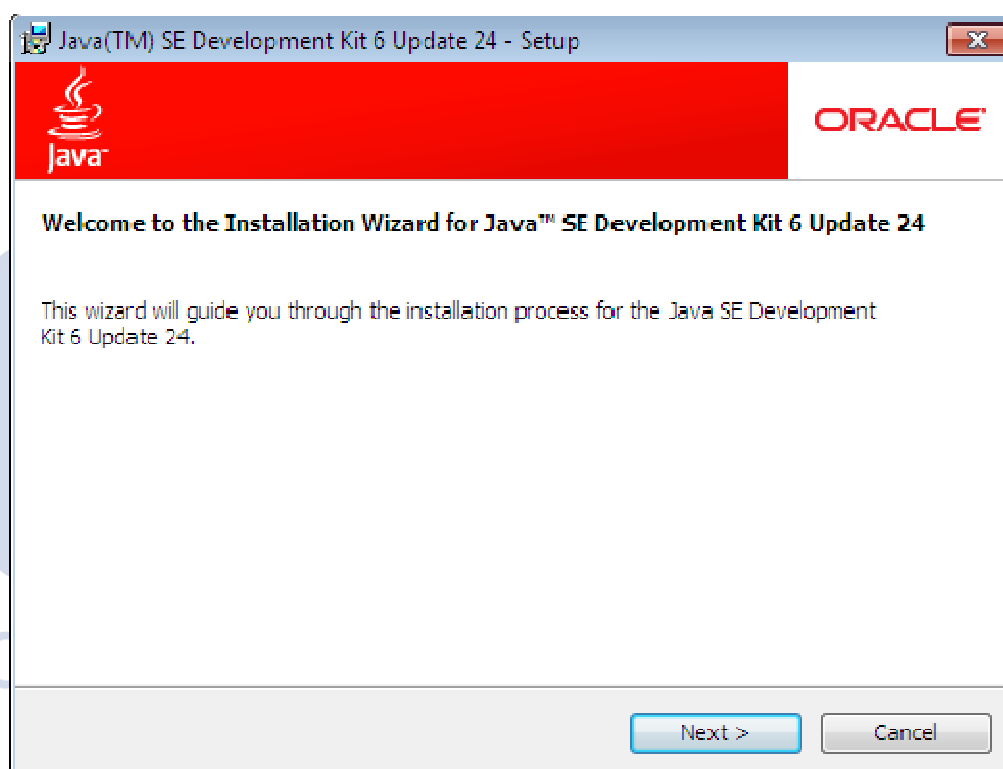
O universo Java é um vasto conjunto de tecnologias, composto por três plataformas principais que foram criadas para segmentos específicos de aplicações:

- **Java SE (Java Platform, Standard Edition)**. É a base da plataforma; inclui o ambiente de execução e as bibliotecas comuns.
- **Java EE (Java Platform, Enterprise Edition)**. A edição voltada para o desenvolvimento de aplicações corporativas e para internet.

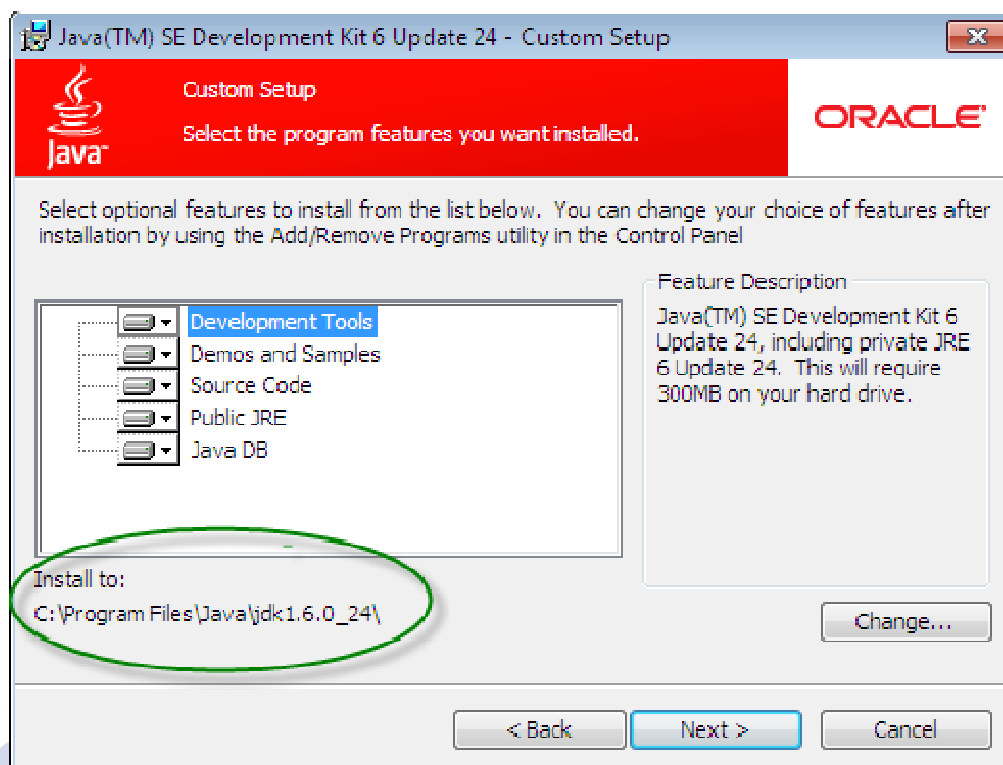
- Java ME (Java Platform, Micro Edition). A edição para o desenvolvimento de aplicações para dispositivos móveis e embarcados.

Instalando o JDK

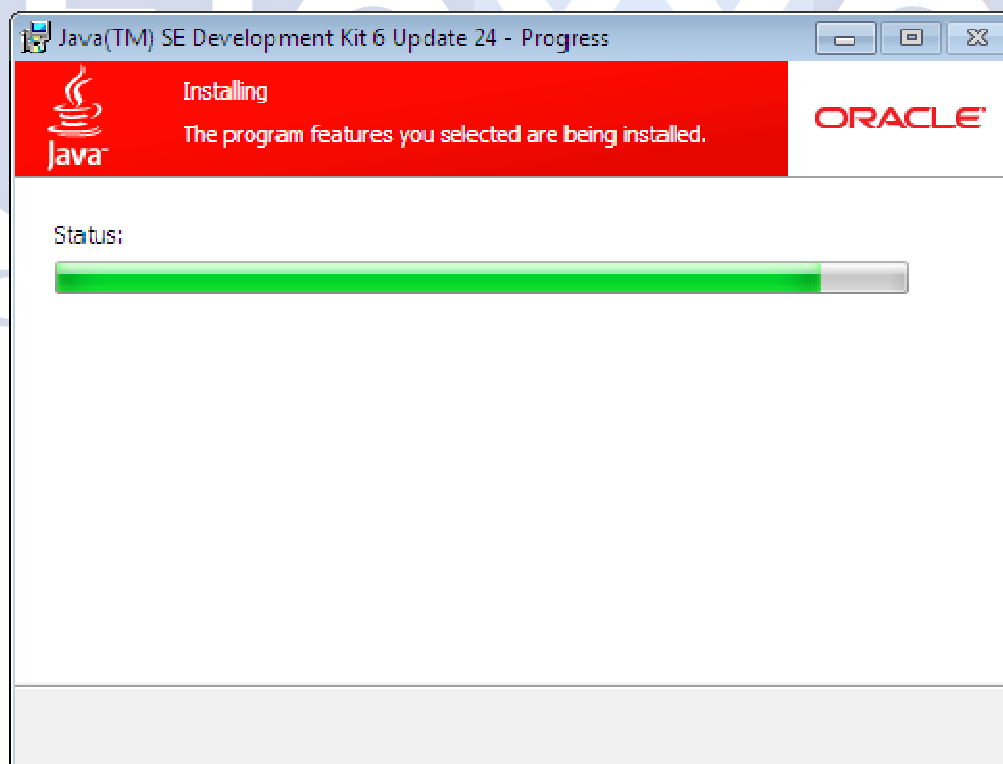
Para instalar a JDK (Java Development Kit), basta baixar o instalador no site da ORACLE (www.oracle.com), seção downloads, menu Java for developers, a seguir clique em Java e selecione a plataforma (sistema operacional) adequado. Após isso basta rodar o executável e seguir os passos da instalação, descritos abaixo.



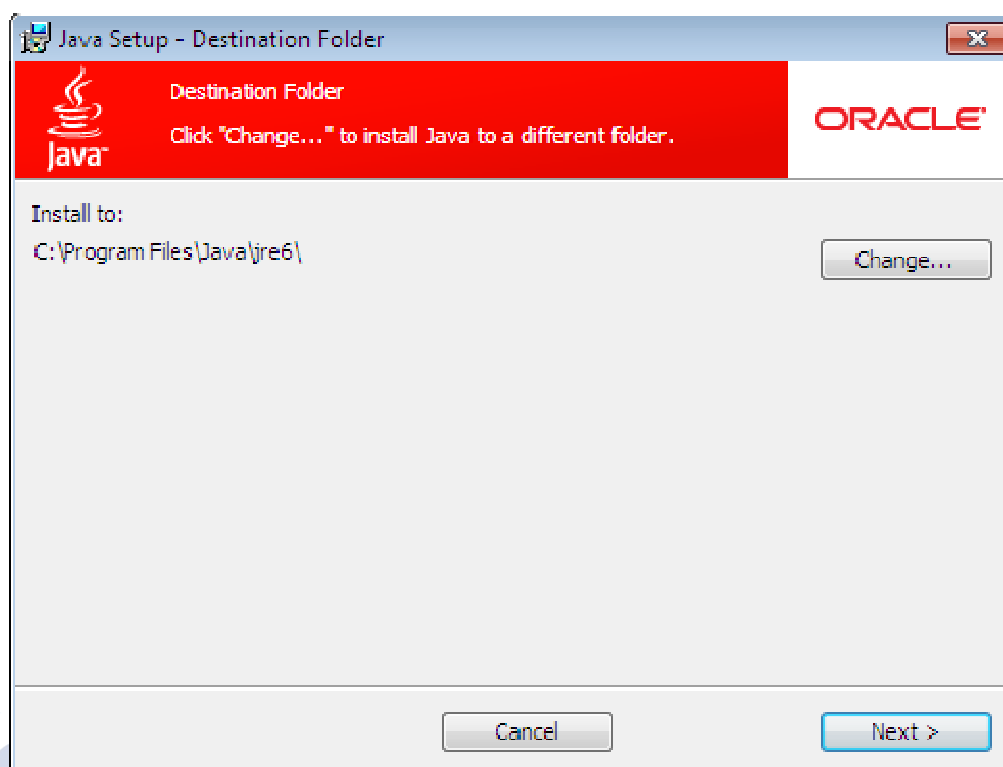
Passo 1: Execute o instalador e nesta tela clique em next



Passo 2: Confira o local de instalação do JDK



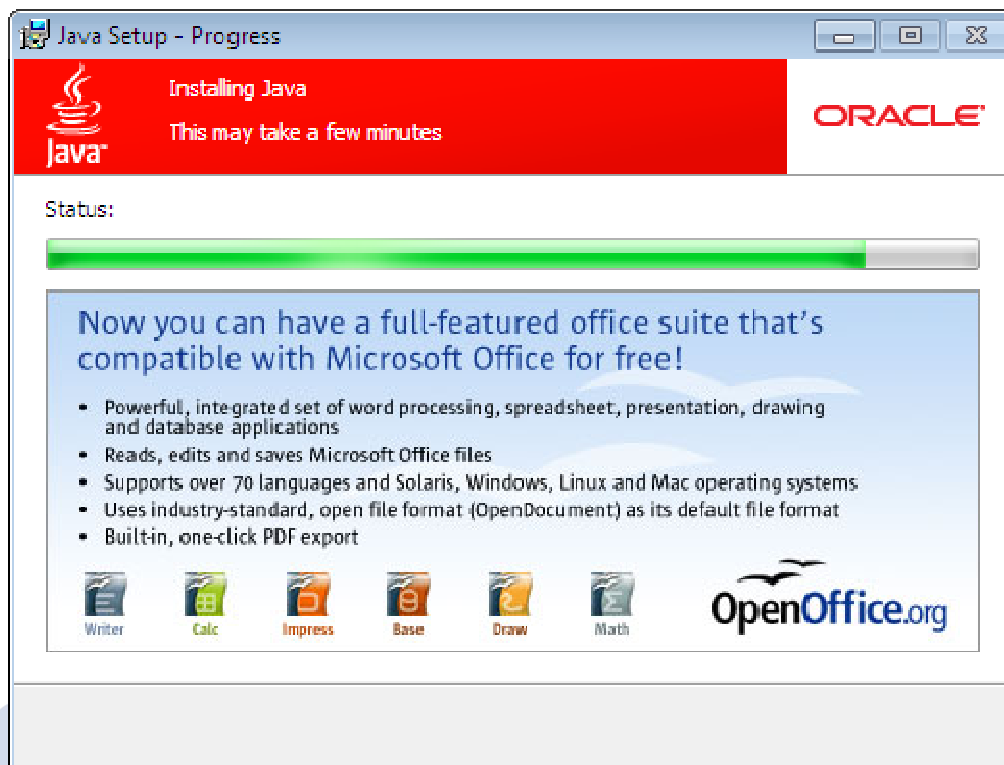
Passo 3: Aguarde a finalização do processo.



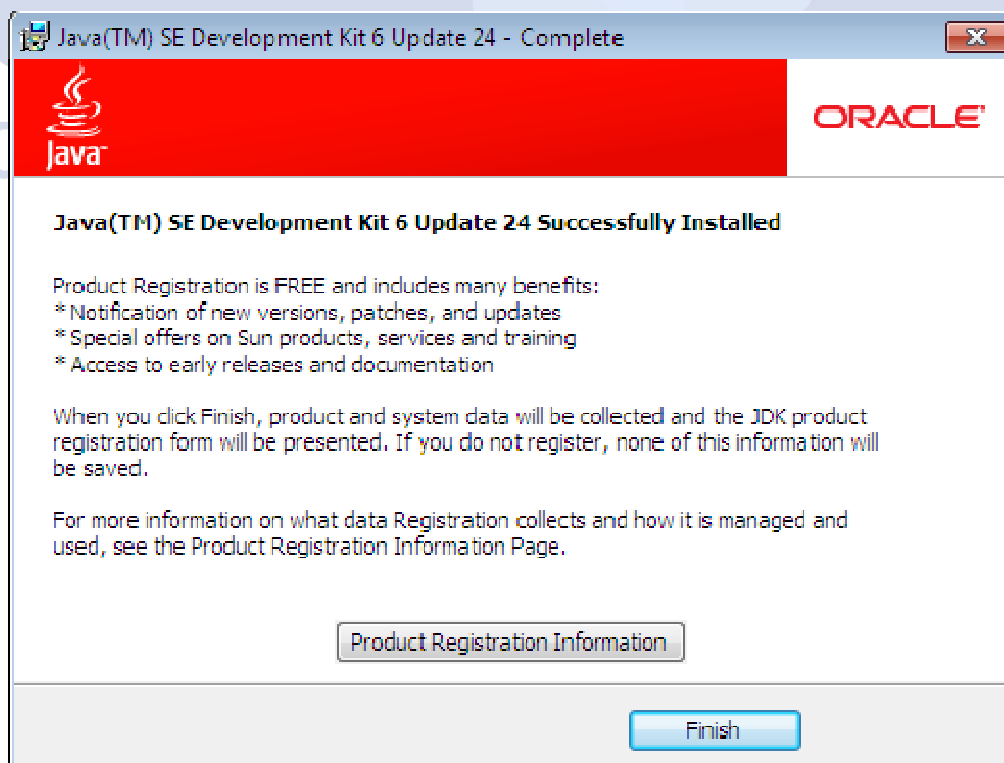
Passo 4: Esta parte refere-se a instalação da jre, utilize a default e clique em next.

Flexxo

Centro de Capacitação em TI

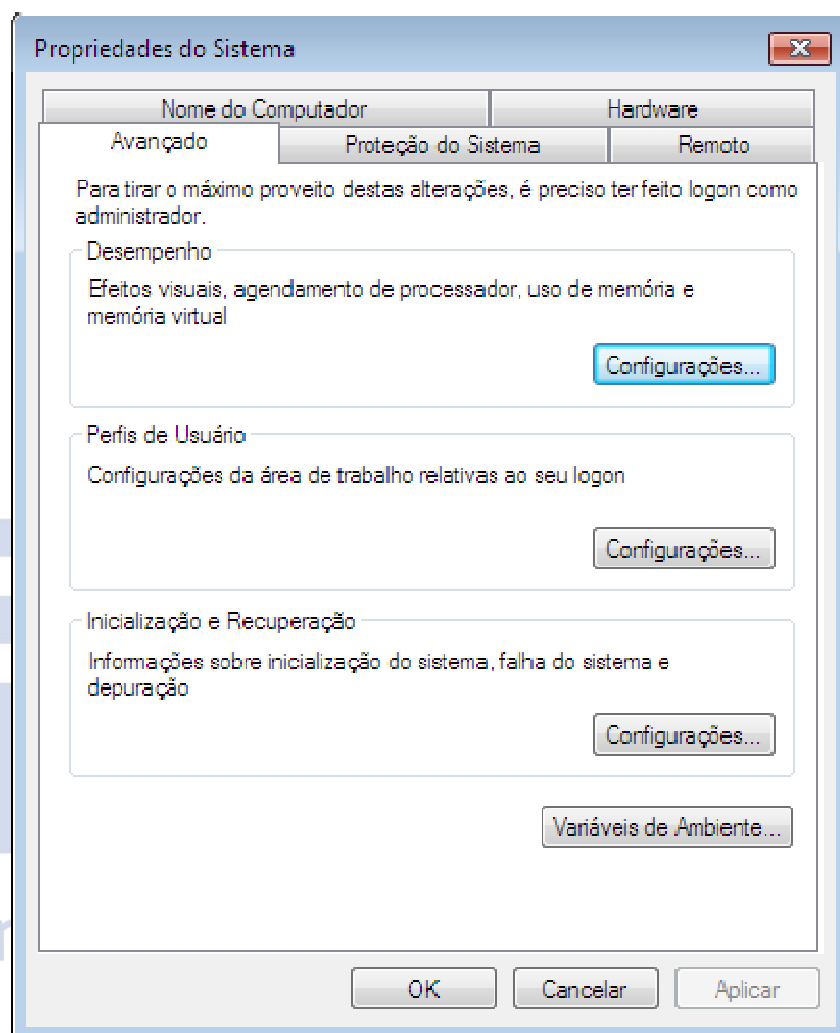


Passo 5: Aguarde o término da instalação da jre

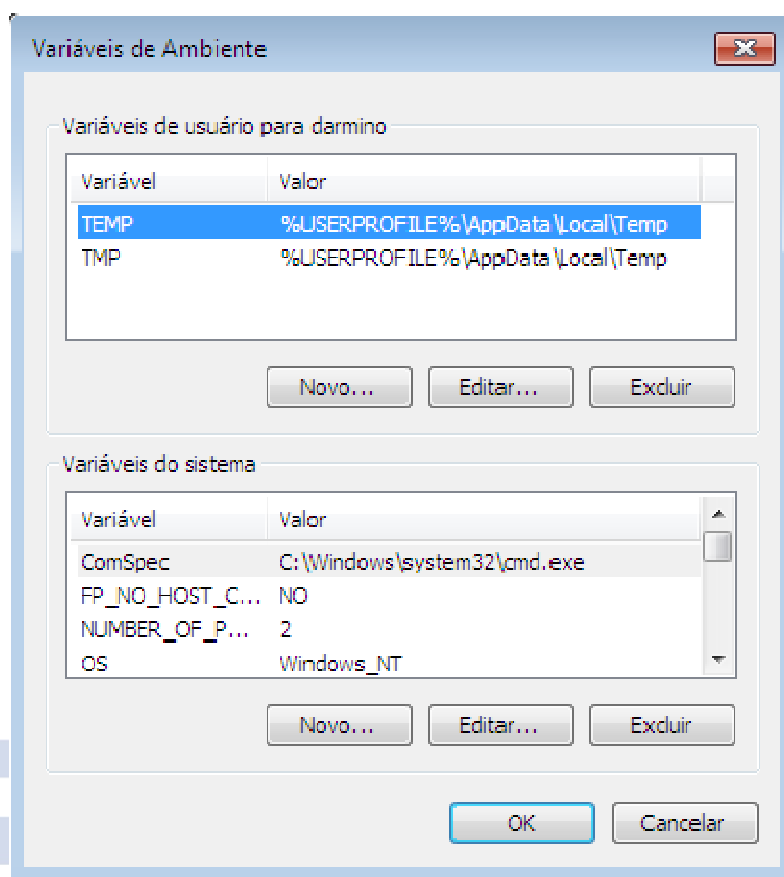


Passo 6: Instalação concluída clique em Finish.

Após instalar o JDK, precisamos setar duas variáveis de ambiente (JAVA_HOME e Path) . Para isso temos que entrar em, Iniciar -> Painel de Controle -> Sistema -> Configurações avançadas, clique em variáveis de ambiente.

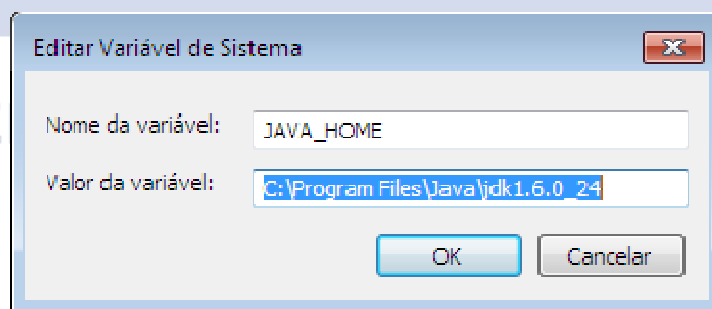


Após isso, verifique se já existe uma variável chamada JAVA_HOME, caso não exista, crie uma variável, clicando em Novo em variáveis do sistema.

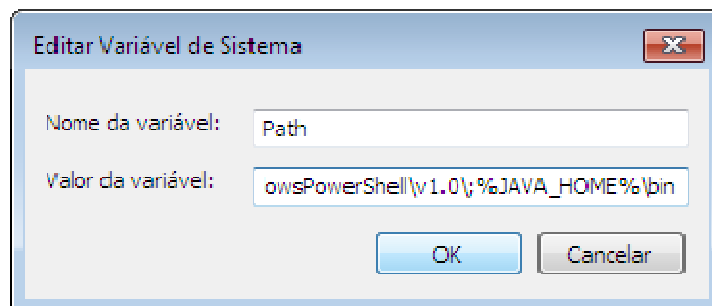


Clique em Novo em variáveis do sistema.

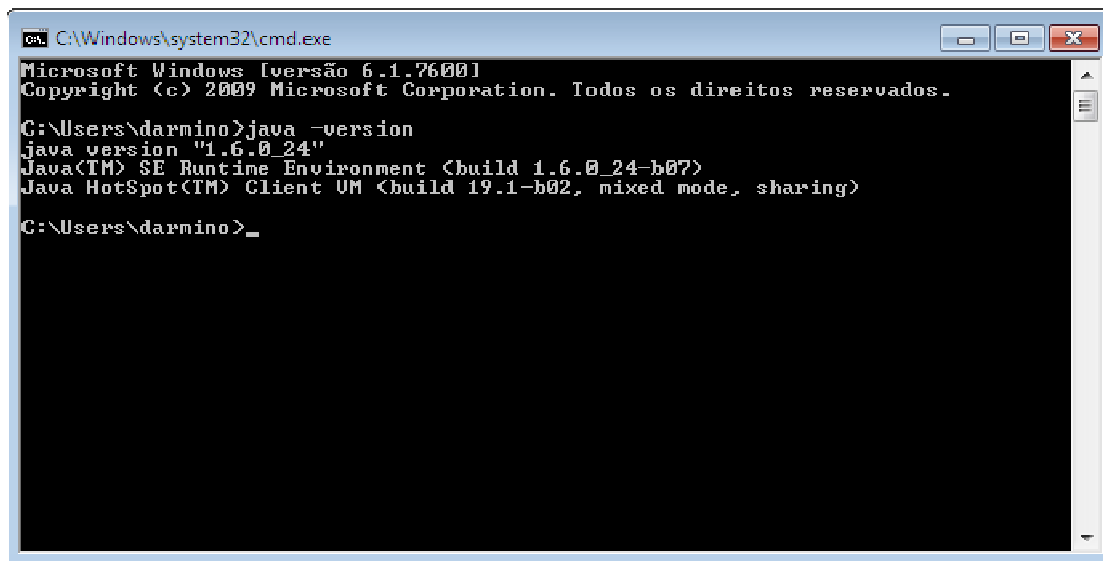
Informe o nome da variável `JAVA_HOME` e no valor coloque o da instalação do JDK.



Após isso, temos que incluir esta variável recém criada na variável Path. Para isso selecione a variável e clique em Editar, vá até o final do valor da variável e concatene -> `;%JAVA_HOME%\bin`



Agora é só testar se tudo funcionou. Para isso abra um prompt de comando e digite `java -version`. O resultado deve ser algo semelhante com a imagem abaixo.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [versão 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Todos os direitos reservados.

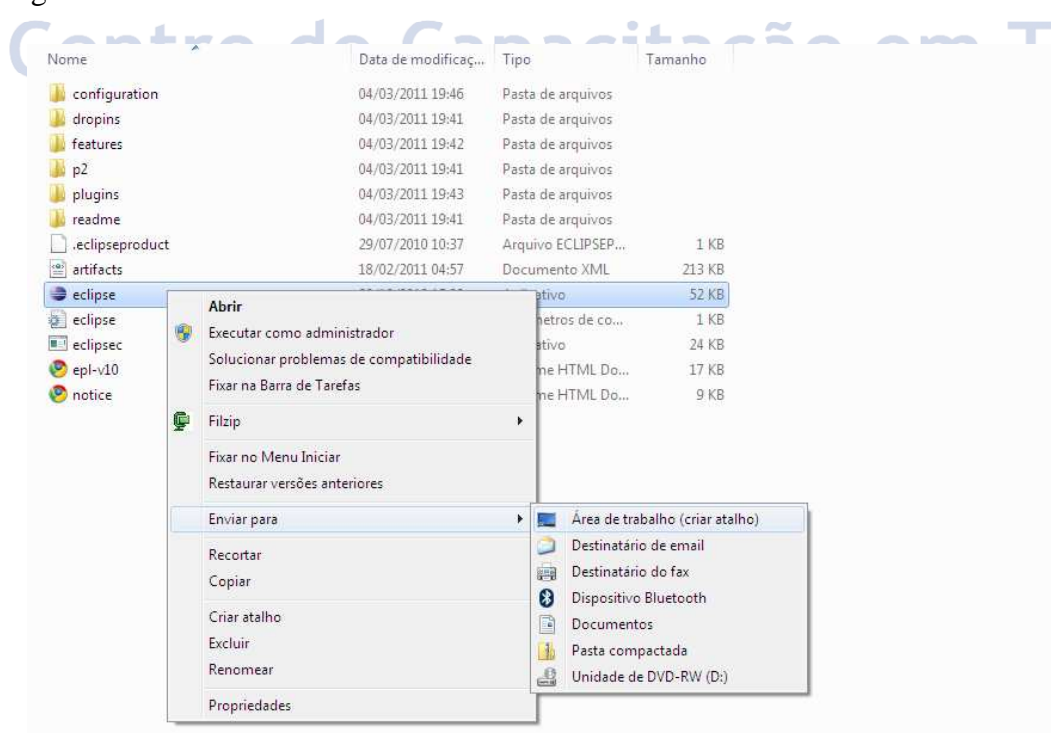
G:\Users\darmino>java -version
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) Client VM (build 19.1-b02, mixed mode, sharing)

G:\Users\darmino>_
```

Se estas mensagens aparecerem para você é porque a instalação foi bem sucedida.

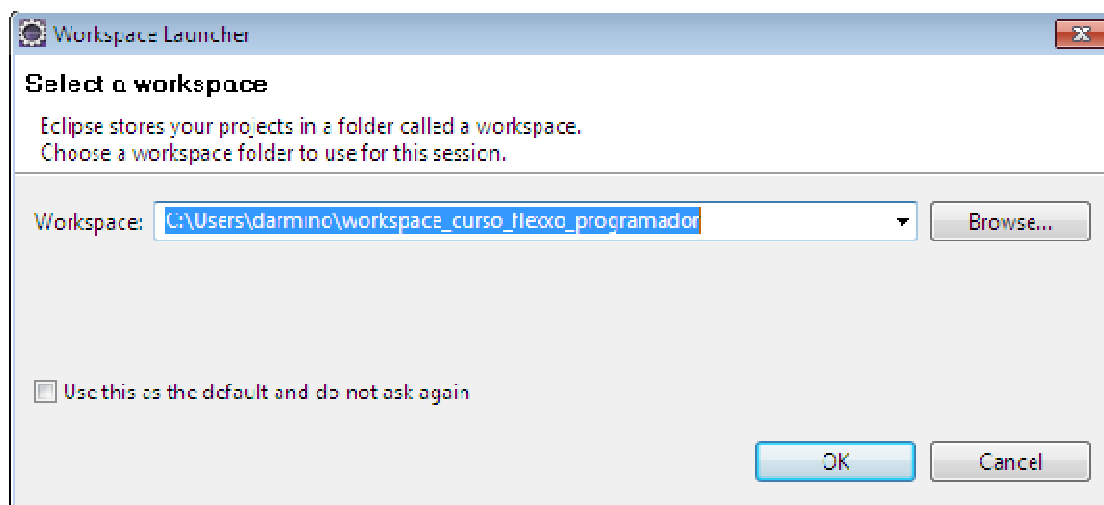
Instalando o eclipse

Para iniciarmos o desenvolvimento, iremos necessitar de uma IDE (Integrated Development Environment) que nos fornecerá uma série de facilidades para o desenvolvimento. Para isso necessitamos efetuar o download da ferramenta no site <http://www.eclipse.org>, acesse o menu downloads e clique em Eclipse IDE for Java EE Developers. Após o download, descompacte e mova a pasta para `c:\Arquivos de Programas`.

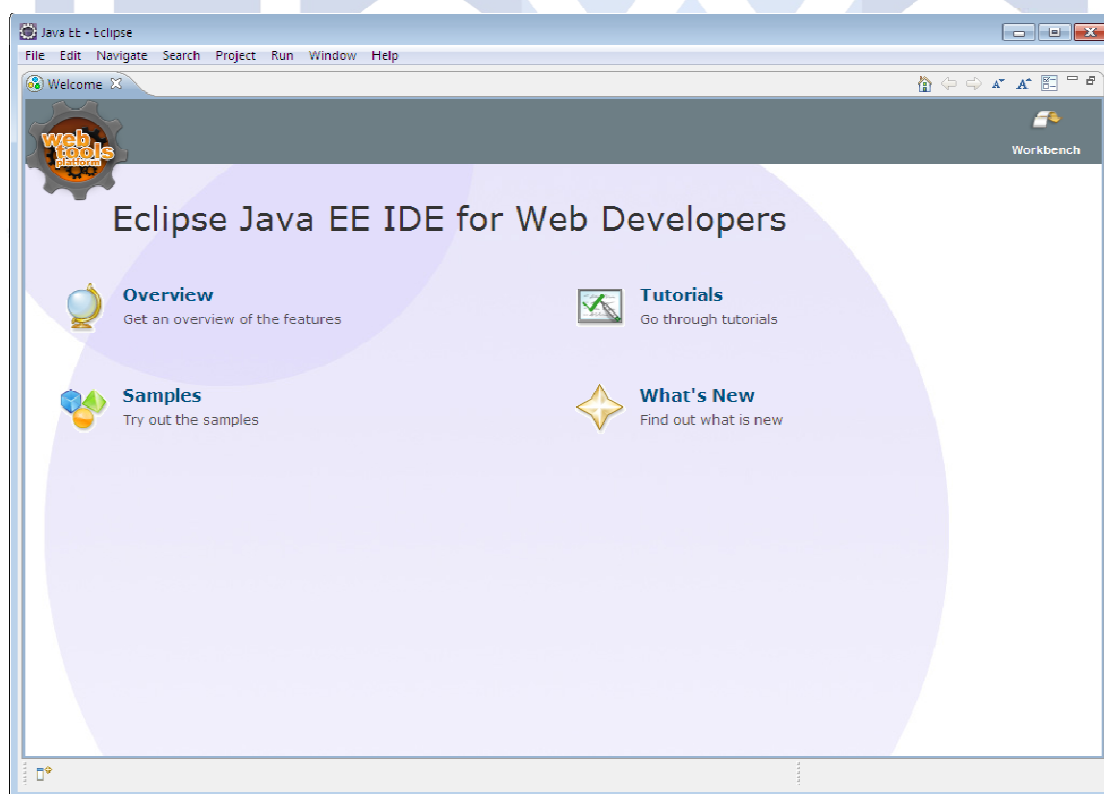


Para facilitar a utilização crie um atalho no desktop.

Após isso podemos executar o programa executando o arquivo que está dentro da pasta chamado eclipse.exe. Devemos fornecer o caminho para o workspace desejado e clicar em ok, todos os arquivos que criarmos ficarão na pasta informada.

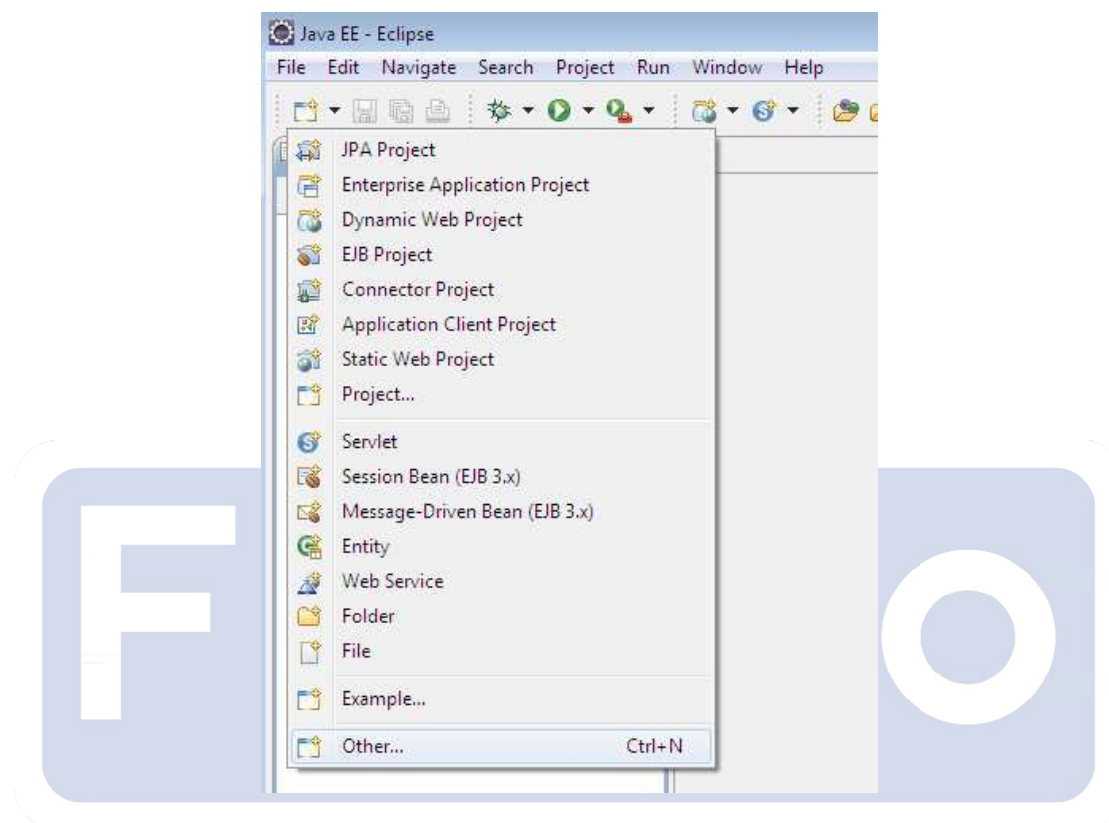


E pronto, o eclipse estará pronto para ser utilizado.

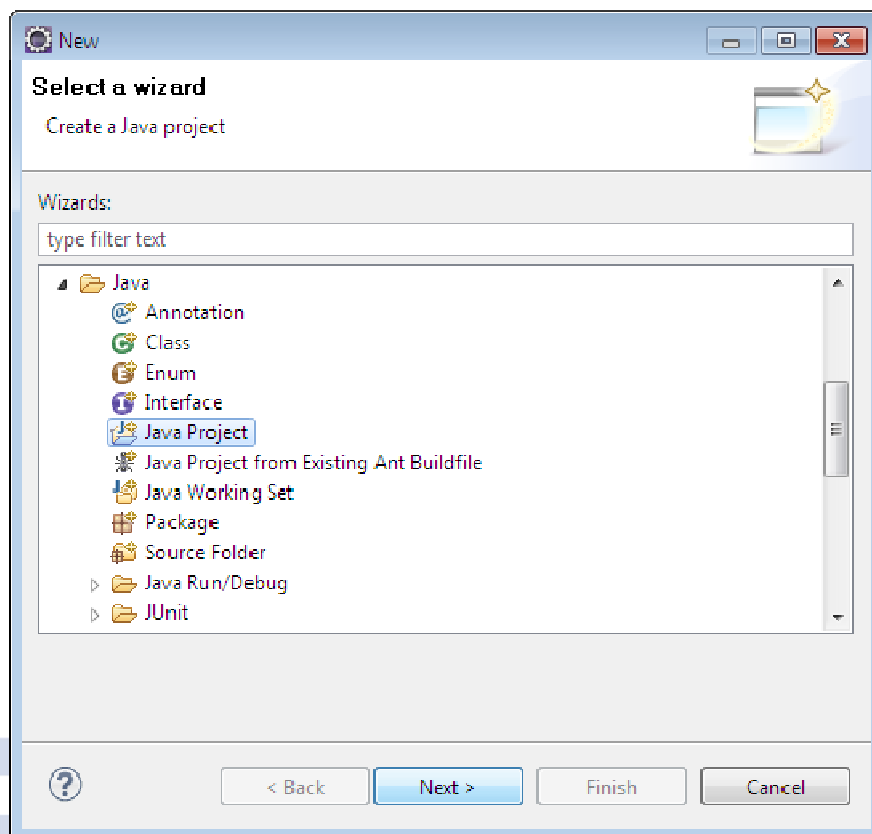


Imprimindo HelloWorld

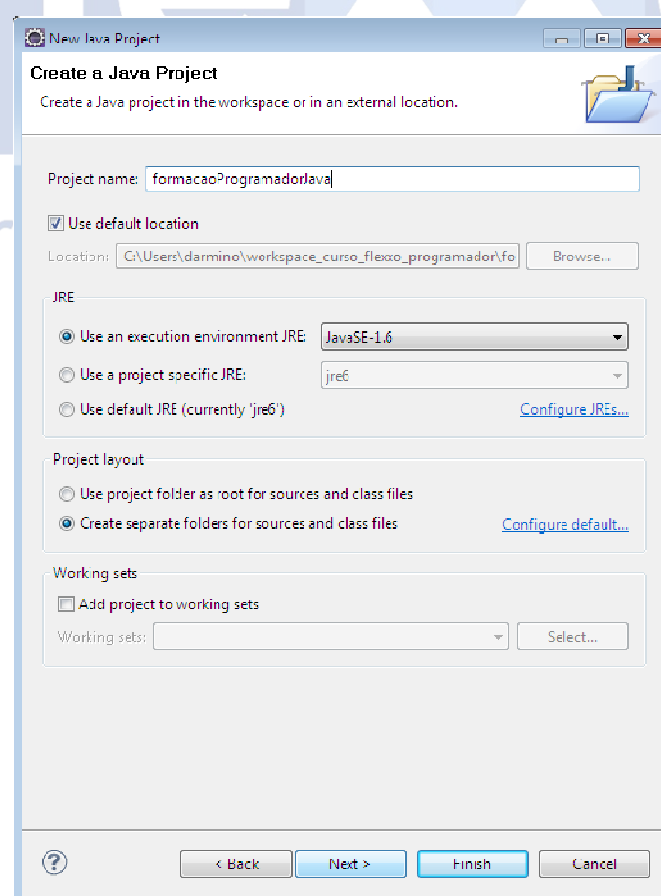
Agora podemos criar uma classe e executá-la para ver como funciona a estrutura básica de um programa Java. Iniciaremos criando um projeto chamado `formacaoProgramadorJava`. Para isso clique na seta do botão New e após em Other.



Centro de Capacitação em TI

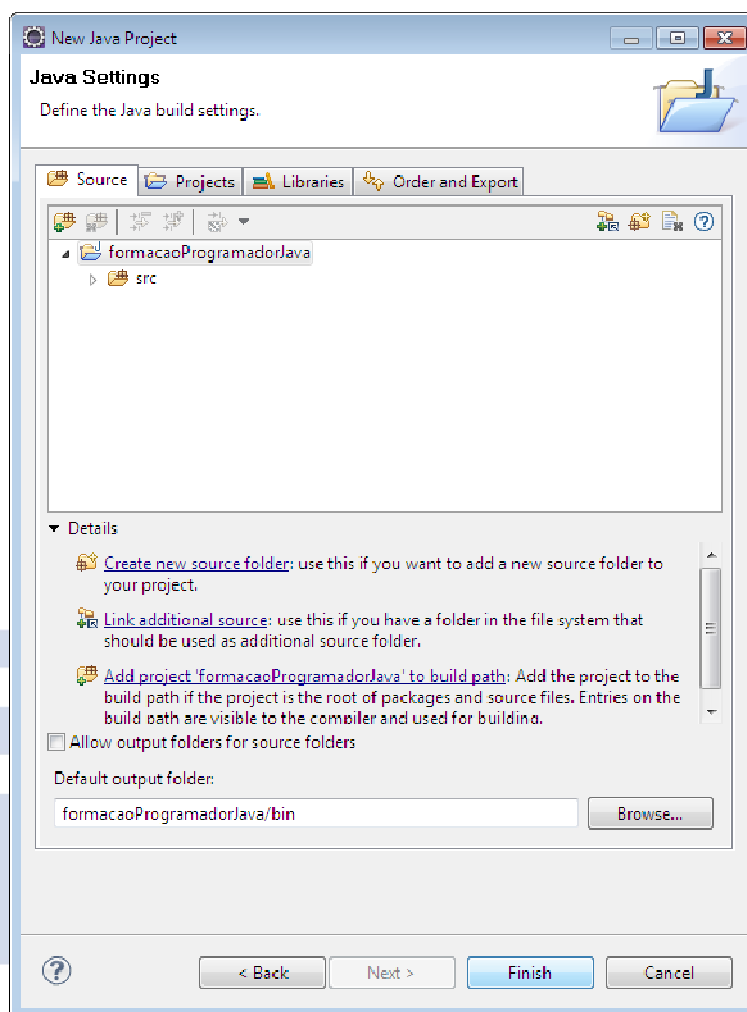


Selecione Java Project e clique em Next.

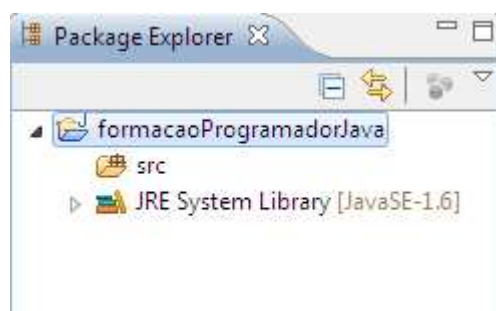


Informe o nome do projeto e clique em Next.

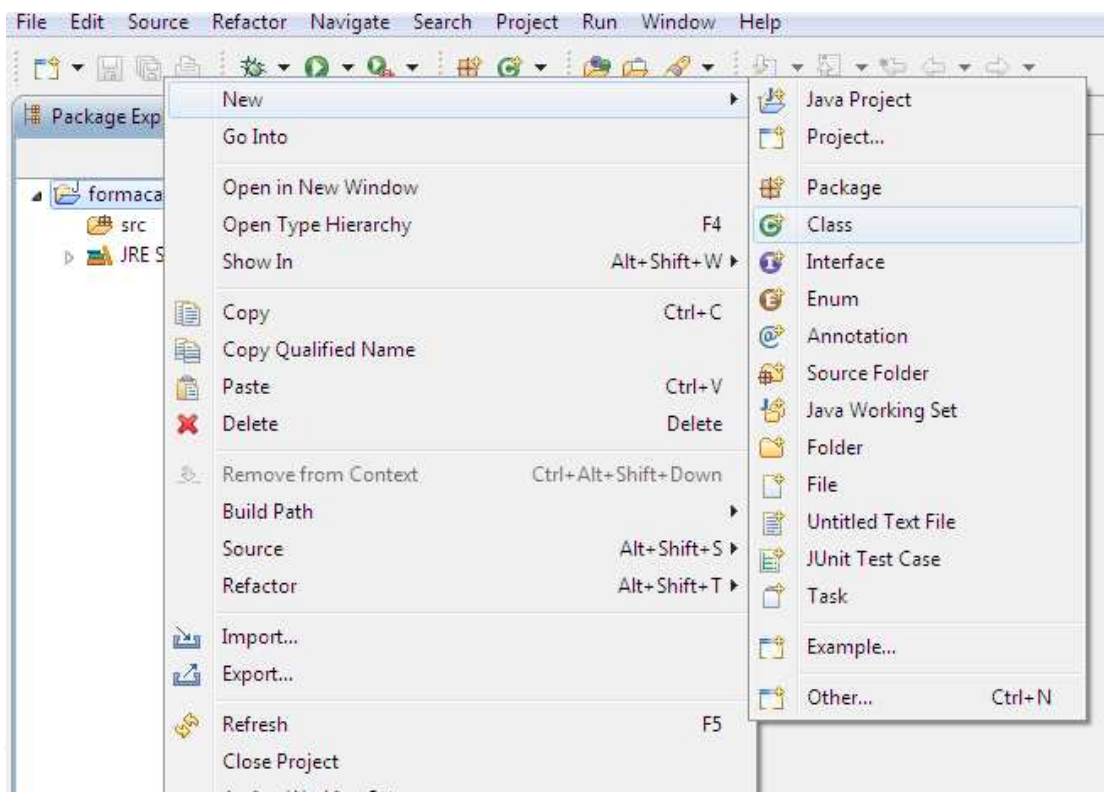
Em seguida será aberta uma janela de configurações do projeto, por ora não precisaremos alterar nada nesta janela, desta forma basta clicar em finish.



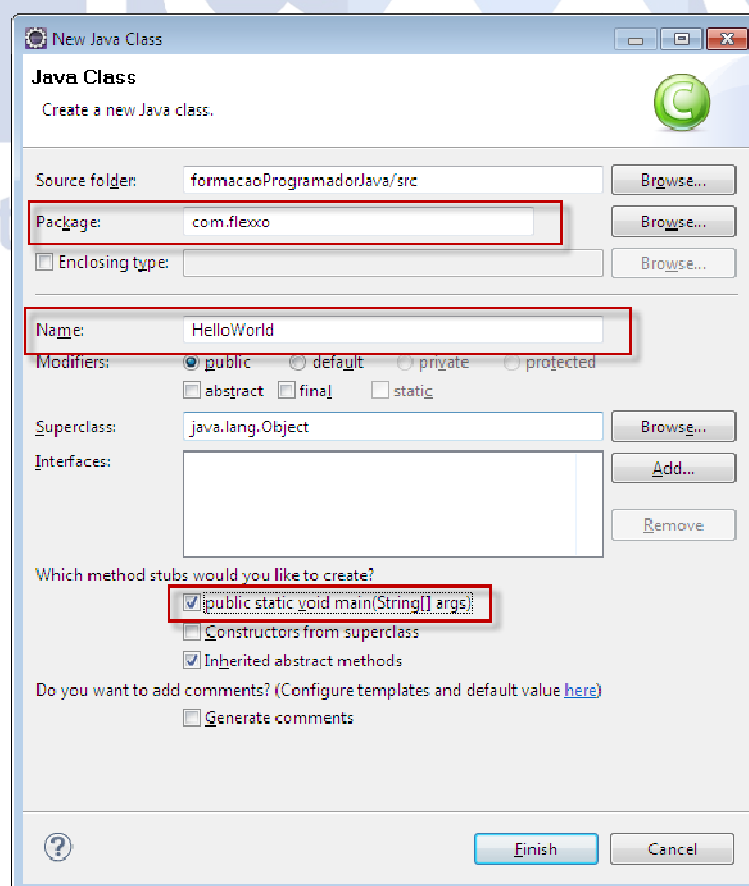
E o projeto estará criado, agora já podemos criar algumas classes Java.



Vamos criar uma classe chamado HelloWorld para começarmos nossos estudos sobre a estrutura básica da linguagem. Para isso clique com o botão direito em cima do projeto e selecione New..., em seguida Java Class...

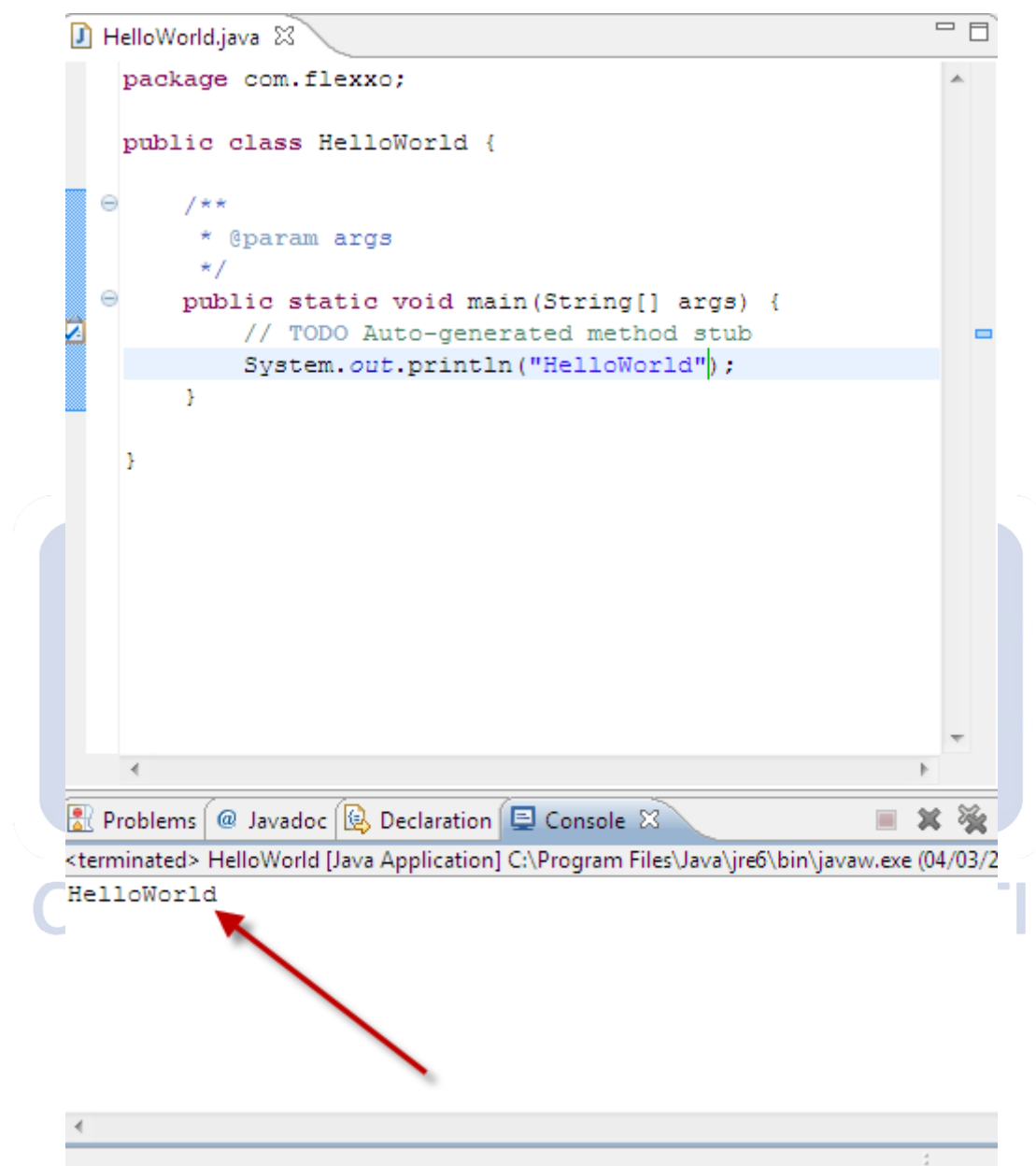


Nesta nova janela que irá abrir informe o package com.flexxo, o nome da classe HelloWorld e selecione o checkbox public static void main(String args[]).



Digite o seguinte comando dentro do método main e após pressione Ctrl +F11.
`System.out.println("HelloWorld");`

A palavra HelloWorld deverá aparecer para você na aba console.



Princípios básicos

Exemplo 1 (arquivo Exemplo1.java):

```

package com.flexxo;

import java.io.Serializable;

public class Exemplo1 implements Serializable {

    private String mensagem = "Helloworld2";

    public void imprimeMsg(String mensagem){
        //Imprime o texto do parâmetro no console
        System.out.println(mensagem);
    }

    public static void main(String args []){
        Exemplo1 exemplo1 = new Exemplo1();
        exemplo1.imprimeMsg("Helloworld");
        exemplo1.imprimeMsg(exemplo1.mensagem);
    }
}

```

The diagram includes the following annotations:

- Pacote da classe**: points to the `package com.flexxo;` line.
- Imports das classes necessárias**: points to the `import java.io.Serializable;` line.
- Modificador de acesso**: points to the `public` keyword in `public class`.
- Nome da classe**: points to the `Exemplo1` in `public class Exemplo1`.
- Variável de instância**: points to the `private String mensagem = "Helloworld2";` line.
- Método da classe**: points to the `public void imprimeMsg` block.
- Método estático main**: points to the `public static void main` block.

Membros de classe

As variáveis de instância formam os atributos de um objeto e, juntamente com os métodos, são os elementos básicos para a formação de uma classe. Eles são denominados membros da classe. Os atributos são espaços em memória reservados para armazenar informações durante a execução da classe. Eles constituem o estado interno de um objeto. Os atributos são inicializados no início da execução da classe e ficam disponíveis para utilização por todos os seus métodos. Os métodos por sua vez definem as operações que podem ser realizadas pela classe.

Método main

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

O método `main` é um método especial pois representa o ponto de entrada para a execução de um programa em Java. Quando um programa é executado, o

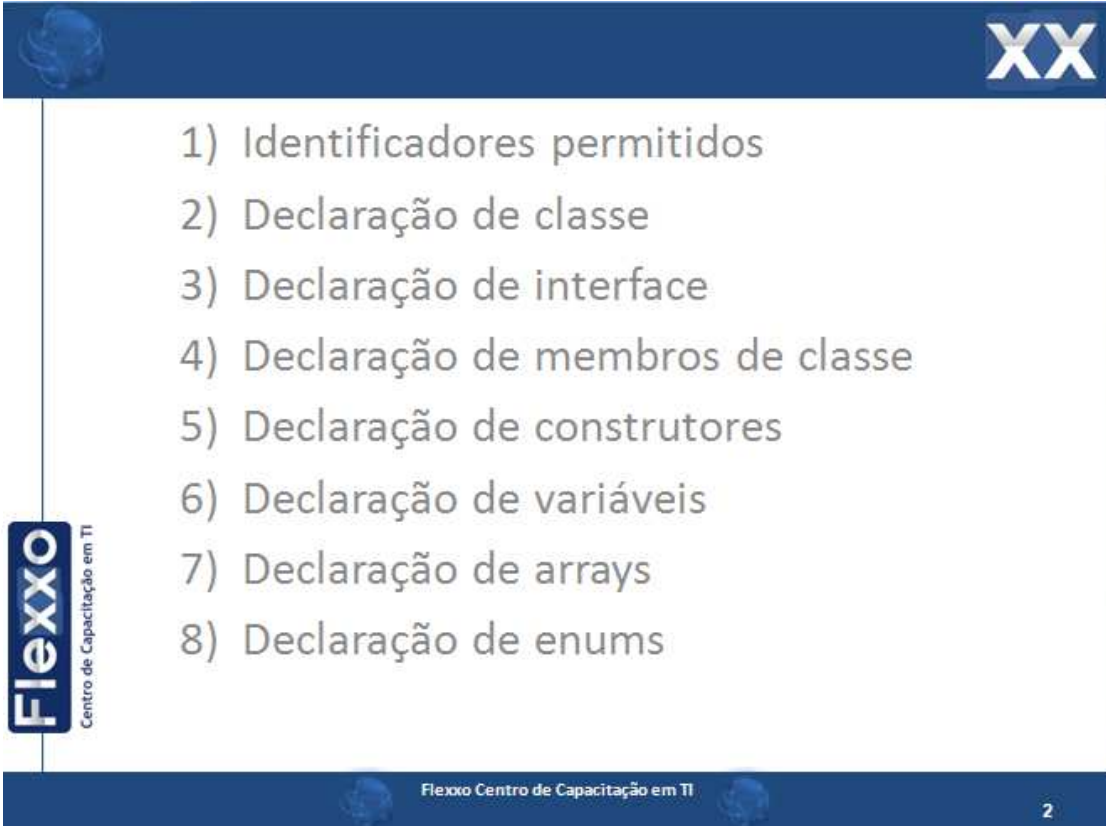
interpretador chamará primeiramente o método main da classe. É ele quem controla o fluxo de execução do programa e executa qualquer outro método necessário para a funcionalidade da aplicação.

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



Declaração e controle de acesso

Neste capítulo veremos:



- 1) Identificadores permitidos
- 2) Declaração de classe
- 3) Declaração de interface
- 4) Declaração de membros de classe
- 5) Declaração de construtores
- 6) Declaração de variáveis
- 7) Declaração de arrays
- 8) Declaração de enums

Regras para identificadores

Os identificadores, isto é, nome das variáveis, classes ou métodos, podem começar com um underscore (_), uma letra ou um cifrão (\$). Depois do primeiro caracter os identificadores podem incluir também dígitos, além de poderem ter qualquer tamanho.

Exemplo de nomes permitidos:

```
private String _nome = null;  
private String nome = null;  
private String $nome = null;
```

Exemplo de nomes não permitidos:

```
private String :nome = null;  
private String _d = null;  
private String e# = null;  
private String .f = null;  
private String 7g = null;
```

Convenções de código

- **Classes e Interfaces:** primeira letra deve ser maiúscula e a primeira letra de cada palavra subsequente deverá ser maiúscula (padrão camelCase). Para classes os nomes geralmente devem ser substantivos. Para interfaces, adjetivos.
Exemplo de nomes de classe: Pedido, Estoque.
Exemplo de interface: Executavel, Serializavel.
- **Métodos:** primeira letra deve ser minúscula e depois a regra camelCase normal. Os métodos normalmente devem ser pares de verbos-substantivo.
Exemplo: buscarRegistros, executaProcesso, somaValores.
- **Variáveis:** idem ao padrão de nomenclatura do método. Usar nomes curtos e significativos.
Exemplo: larguraDoBotao, contadorDeRegistros...
- **Constantes:** Constantes são criadas com os modificadores static final. Elas devem ser nomeadas usando-se letra maiúscula com underscore entre as palavras.
Exemplo: ALTURA_MINIMA.

Padrões JavaBeans

A especificação JavaBeans foi criada com o objetivo de padronizar a nomenclatura de métodos e variáveis, auxiliando assim desenvolvedores a identificar melhor métodos já existentes (possivelmente desenvolvido por terceiros) para sua necessidade, além de determinar a padronização e facilitar que ferramentas de desenvolvimento (IDE'S) auxiliem o desenvolvedor em determinadas tarefas.

As regras são as seguintes:

- 1) Todo a variável de instância deve ser privada e possuir dois métodos relacionados, um para retornar o valor (get) e outro para setá-lo (set).
- 2) As assinaturas de um método set e get deve obedecer o seguinte exemplo:

```
private String mensagem = null;

public String getMensagem() {
    return mensagem;
}

public void setMensagem(String mensagem) {
    this.mensagem = mensagem;
}
```

- 3) Caso o tipo da variável ser booleano, ao invés do get pode ser usado is, também.
- 4) Para listeners vale o exemplo abaixo:

```
public void addMyListener (MyListener m) {
    // Code here
}
```



```
public void removeMyListener (MyListener m) {
    // Code here
}
```

Declaração de classes

Regras para declaração de classes:

1. Só pode haver uma classe public em cada arquivo de código-fonte.
2. Os comentários podem aparecer em cada em qualquer lugar do arquivo.
3. O nome da classe public deve ser o mesmo que o nome do arquivo.
4. A primeira linha de comando de um arquivo java deve ser a package caso exista.
5. A partir da segunda linha, vem os imports.
6. O modificador de acesso de classe só pode ser public ou default.

Modificadores não referentes a acesso

- **Final:** Quando usada na declaração de uma classe, significa que esta classe não poderá ser estendida, ou seja, nenhuma classe poderá herdá-la.
- **Abstract:** Uma classe abstract é justamente o contrário da Final, ou seja, indica que a classe deve ser estendida, visto que classes abstract não podem ser instanciadas.

Exemplo:

```
public class Exemplo1 extends ExemploFilha implements Serializable
{
    //code here
}
```

The diagram labels the following parts of the code:

- public ou default:** Points to the `public` keyword.
- abstract ou final (não obrigatório):** Points to the `abstract` or `final` keywords.
- Nome da classe:** Points to the class name `Exemplo1`.
- Classe à estender:** Points to the `ExemploFilha` class being extended.
- Interface à implementar:** Points to the `Serializable` interface being implemented.

Declaração de interfaces

Regras para declaração de interfaces:

1. Todos os métodos de uma interface são implicitamente public e abstract, mesmo que não sejam declarados na interface o compilador irá colocar estes modificadores em tempo de compilação.
2. Todas as variáveis de uma interface são implicitamente public, static e final, ou seja, uma interface só pode ter constantes.
3. Métodos em interface não podem ser static, final, native, strictfp ou synchronized.
4. Uma interface pode estender uma ou mais interfaces.
5. Uma interface não implementa nada.

Exemplo:

```
public interface InterfaceExemplo extends InterfaceExemplo2
{
}
}
```

Declaração de membros de classe

O que são membros de classe ? Os métodos e variáveis de instância são conhecidos como membros de classe. Todos os modificadores de acesso possuem um nível de visibilidade, que são os chamados modificadores de acesso. É o modificador de acesso que irá determinar em que classes estes membros poderão ser utilizados.

Modificadores de acesso

Os modificadores de acesso podem ser de 4 tipos:

- **public:** Qualquer classe poderá acessar um método ou variável declarado com este modificador.
- **protected:** Apenas classes do mesmo pacote e classes filhas (independente do pacote) podem acessar um membro com esta visibilidade.
- **private :** Membros declarados com este modificador só podem ser utilizados dentro da própria classe.
- **default:** Apenas classes do mesmo pacote podem acessar um membro com esta visibilidade.

Modificadores não referentes à acesso

Existem outros tipos de modificadores, são eles:

- **final :** para métodos indica que estes não podem ser sobrescritos nas classes filhas, já para variáveis, indica que a variável nunca poderá ter o seu valor modificado.
- **abstract:** utilizado em métodos, indica que o método deverá ser sobrescrito na primeira classe concreta que estender a classe na qual ele se encontra. Quando algum método de classe possuir este identificador então será obrigatório que este esteja na declaração da classe. Além disso, métodos declarados com abstract, não possuem corpo.

Exemplo:

```
public abstract void setMensagem(String mensagem);
```

- **synchronized:** Este modificador é aplicado apenas a métodos e garante que apenas uma thread estará executando este método por vez, ou seja, nunca acontecerá uma execução simultânea de duas threads para o método marcado com este modificador.
- **native:** Aplicado apenas em métodos, indica que o método está sendo implementado conforme a plataforma (Windows ou Linux), geralmente estes métodos são escritos em C.
- **strictfp:** aplicado em classes e métodos, força os pontos flutuantes a aderirem o padrão IEEE 754.
- **volatile:** O modificador volatile diz à JVM que um thread que acesse a variável deve sempre reconciliar a sua cópia private com a cópia master presente na memória.
- **transient:** Variáveis marcadas com este modificador não serão submetidas no processo de serialização/deserialização de objetos.
- **static:** este modificador pode ser aplicado tanto para método como variáveis e indica que o valor das variáveis será apenas 1 para todas as instancias da classe. Com este modificador é possível acessar qualquer membro de classe sem ter uma instancia da classe.

Declaração de construtores de classe

Em Java, os objetos são construídos. Sempre que um objeto é criado, seu construtor é invocado. Toda classe tem um construtor e mesmo que não exista um na classe, o compilador criará um implicitamente. Construtores são semelhantes aos métodos porém com 2 diferenças: o nome do construtor deve ser o mesmo da classe e o construtor não tem retorno.

Exemplo:

```
public class HelloWorld {  
  
    public HelloWorld(){  
        //codeHere  
    }  
}
```

Variáveis

Existem dois tipos de variáveis em Java:

- **Primitivas:** Um primitivo pode ser de oito tipos: char, boolean, byte, short, int, long, double ou float.
- **Referência:** são variáveis usadas para se referir a objetos. Uma variável de referência é declarada como um tipo específico e este tipo nunca pode ser modificado.

Intervalo e tamanhos de primitivos

Abaixo uma tabela contendo os tipos primitivos com seus respectivos tamanhos e intervalos.

Tipo	Bits	Bytes	Interv. Min.	Interv. Max.
byte	8	1	-2^7	$2^7 - 1$
short	16	2	-2^{15}	$2^{15} - 1$
int	32	4	-2^{31}	$2^{31} - 1$
long	64	8	-2^{63}	$2^{63} - 1$
float	32	4	-2^{31}	$2^{31} - 1$
double	64	8	-2^{63}	$2^{63} - 1$
boolean	-	-	-	-
char	16 (unsigned)	2	0	65535

boolean: este tipo de dado possui apenas 2 valores (true ou false).

Declaração de arrays

Em java, os arrays são objetos que armazenam múltiplas variáveis do mesmo tipo, ou variáveis que são todas do mesmo tipo. Os arrays podem armazenar ou primitivos ou referência a objetos, mas o próprio array será sempre um objeto no heap.

Exemplo:

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

Declaração de enums

A partir da versão 5.0, Java lhe permite restringir uma variável para ter um apenas alguns valores pré-definidos.

Exemplo:

```
package com.flexxo;

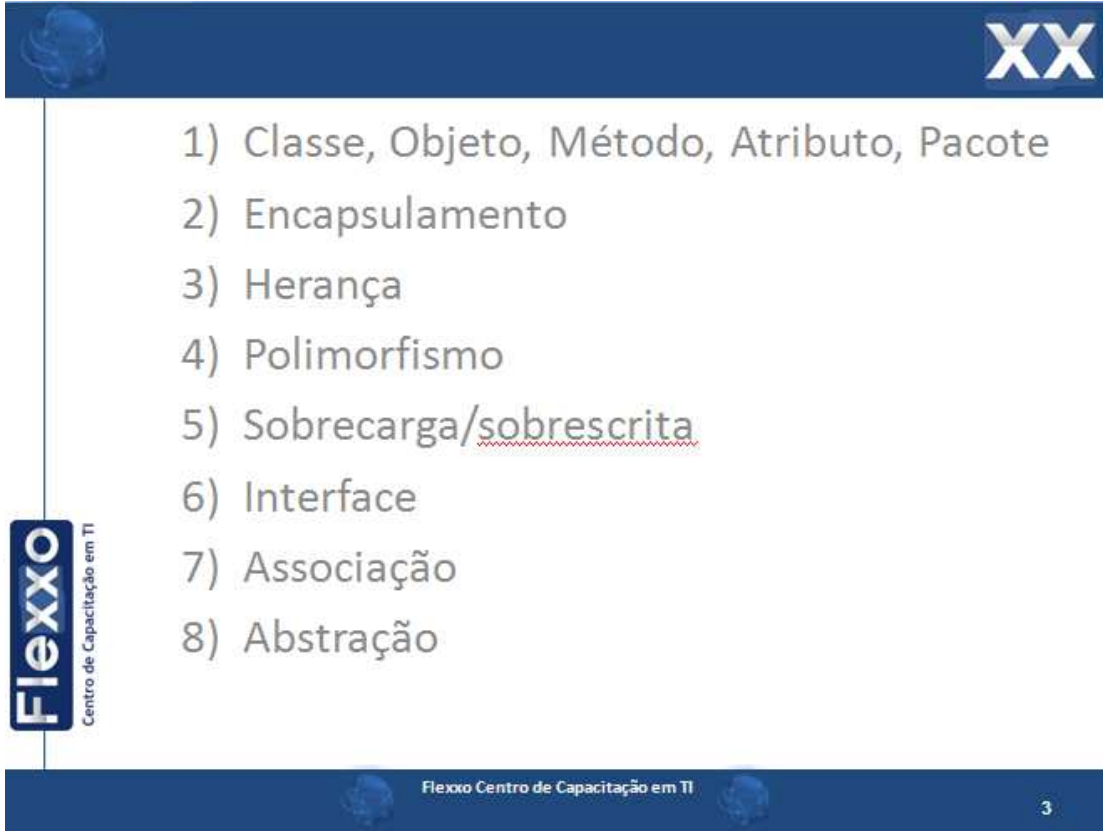
enum TamanhoDoCafe{ PEQUENO, MEDIO, GRANDE}

public class ExemploEnum {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        TamanhoDoCafe a = TamanhoDoCafe.GRANDE;
        System.out.println(a);
    }

}
```

Orientação a objetos



1) Classe, Objeto, Método, Atributo, Pacote
2) Encapsulamento
3) Herança
4) Polimorfismo
5) Sobrecarga/sobrescrita
6) Interface
7) Associação
8) Abstração

Flexxo
Centro de Capacitação em TI

Flexxo Centro de Capacitação em TI 3

Classe

Representa um conjunto de objetos com características afins. Uma classe define o comportamento dos objetos através de seus métodos, e quais estados ele é capaz de manter através de seus atributos. É a “fôrma” dos objetos. Exemplo de classe: Os seres humanos.

Objeto

Um objeto é capaz de armazenar estados através de seus atributos e reagir a mensagens enviadas a ele, assim como se relacionar e enviar mensagens a outros objetos. Exemplo de objetos da classe Humanos: João, José, Maria.

Método

Os métodos determinam as “habilidades” dos objetos. Digamos que Bidu é uma instância da classe Cachorro, portanto tem habilidade para latir, implementada através do método “latir”. Um método em uma classe é apenas uma definição. A ação só ocorre quando o método é invocado através do objeto, no caso Bidu. Dentro do programa, a utilização de um método deve afetar apenas um objeto em particular; Todos os cachorros podem latir, mas você quer que apenas Bidu dê o latido.

Normalmente, uma classe possui diversos métodos, que no caso da classe Cachorro poderiam ser sentar, comer e morder.

Atributo

São características de um objeto. Basicamente a estrutura de dados que vai representar a classe. Exemplos: Funcionário: nome, endereço, telefone, CPF,...; Carro: nome, marca, ano, cor, ...; Livro: autor, editora, ano. Por sua vez, os atributos possuem valores. Por exemplo, o atributo cor pode conter o valor azul. O conjunto de valores dos atributos de um determinado objeto é chamado de estado.

Pacotes

São referências para organização lógica de classes e interfaces.

Encapsulamento

Corresponde ao ato de isolar o acesso direto a variáveis de instâncias de uma classe. Cada variável de instância terá um método get e um set e apenas por eles será possível obter (get), ou setar (set) o valor.

Ganho: flexibilidade e facilidade de manutenção de código, visto que a “regra” estará num lugar apenas.

Exemplo com encapsulamento:

```
package com.flexxo;

public class Encapsulamento {

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

    public void setName(String name) {
        this.name = name;
    }

}
```

Aqui poderíamos querer efetuar uma validação qualquer, desta forma apenas em um lugar no sistema teríamos esta validação.

Acessando o atributo:

```
public static void main(String [] args){
    Encapsulamento a = new Encapsulamento();
    a.setName("Daniel");
}
```

Exemplo sem encapsulamento:

```
package com.flexxo;

public class Encapsulamento {
    public String name;
}
```

Qualquer validação ficaria aqui e em todos os lugares no sistema que setarem o valor.

Acessando o atributo:

```
public static void main(String [] args){
    Encapsulamento a = new Encapsulamento();
    a.name = "Daniel";
}
```

Herança

Herança significa herdar algo, neste caso herdar métodos e atributos de outras classes. Isto significa que se tivermos necessidade de criar uma classe idêntica a outra no sistema, porém com um método a mais ou uma variável a mais ou mesmo reescrever um método da classe pai, podemos utilizar herança de classe (extends), para não ter que reescrever toda a classe.

Exemplo:

Classe pai:

```
package com.flexxo;

public class Carro {
    private String cor;

    public void correr(){
        System.out.print(" Aceleracao maxima 95 km/h");
    }
}
```

Classe filha (atributo cor é herdado pela classe):

```
package com.flexxo;

public class Ferrari extends Carro {

    public void correr(){
        System.out.println(" Cor "+ cor +"Aceleracao maxima 290 km/h");
    }
}
```

Polimorfismo

Um dos conceitos mais complicados de se entender, e também um dos mais importantes, é o Polimorfismo. O termo polimorfismo é originário do grego e significa "muitas formas".

Na orientação a objetos, isso significa que um mesmo tipo de objeto, sob certas condições, pode realizar ações diferentes ao receber uma mesma mensagem. Ou seja, apenas olhando o código fonte não sabemos exatamente qual será a ação tomada pelo sistema, sendo que o próprio sistema é quem decide qual método será executado, dependendo do contexto durante a execução do programa.

Exemplo:

Classe pai:

```
package com.flexxo;

public class Carro {

    public void correr(){
        System.out.print(" Aceleracao maxima 95 km/h");
    }
}
```

Classe filha (atributo cor é herdado pela classe):

```
package com.flexxo;

public class Ferrari extends Carro {

    public void correr(){
        System.out.println(" Cor "+ cor +"Aceleracao maxima
290 km/h");
    }
}
```

Se tivéssemos o seguinte trecho, que método seria chamado ?

```
Carro carro = new Ferrari();
carro.correr();
```

Devido ao polimorfismo, o método a ser chamado é o da classe Ferrari.

Sobrecarga/Sobrescrita

Sobrecarga é o ato de escrever um método com o mesmo nome de um outro, porém com parâmetros diferentes.

Sobrescrita é o ato de reescrever métodos da classe pai, afim de alterar seu comportamento.

Exemplo de sobrecarga:

```
public class Ferrari extends Carro {

    public void correr(){
```



```

        System.out.println(" Aceleracao maxima de 80
km/h");
    }

    public void correr(int velocidade){
        System.out.println(" Aceleracao maxima de
        "+velocidade+" km/h");
    }
}

```

Exemplo de sobrescrita:

Classe pai:

```

package com.flexxo;

public class Carro {
    public void correr(){
        System.out.print(" Aceleracao maxima 95 km/h");
    }
}

```

Classe filha (método corer sobrescrito):

```

package com.flexxo;

public class Ferrari extends Carro {
    public void correr(){
        //do code here
    }
}

```

Conversão de variáveis de referência

Existem dois tipos de conversão de variáveis de referência:

Redutora: Conversão refere-se quando convertemos um objeto para baixo na árvore de hierarquia. Exemplo:

```

Carro a = new Ferrari();
Ferrari b = (Ferrari) a;

```

Generalizadora: Ao contrário da anterior, esta ocorre quando convertemos para cima na árvore de hierarquia. Exemplo:

```

Ferrari a = new Ferrari();
Carro b = a;

```

Interfaces

Interface é um resumo dos métodos que a classe que implementá-la deverá implementar. Ou seja, ela obriga todas as classes concretas que a implementarem-na, implementar todos os seus métodos.

Interface

```
package com.flexxo;  
  
public interface Voavel {  
    public void voar();  
}
```

Classe

```
package com.flexxo;  
  
public class Pato implements Voavel {  
    public void voar() {  
        System.out.print("voando");  
    }  
}
```

Associação

É o mecanismo pelo qual um objeto utiliza os recursos de outro. Pode tratar-se de uma associação simples "usa um" ou de um acoplamento "parte de". Por exemplo: Um humano usa um telefone. A tecla "1" é parte de um telefone.

Abstração

É a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Em modelagem orientada a objetos, uma classe é uma abstração de entidades existentes no domínio do sistema de software.

Centro de Capacitação em TI

Atribuições

Literais, atribuições e variáveis

Um literal primitivo é simplesmente a representação do código-fonte dos tipos de dados primitivos, em outras palavras, um inteiro, um número de ponto flutuante, um booleano ou caractere que você digite enquanto escreve um código.

Literais inteiros

Há 3 maneiras de representar números inteiros: decimal (base 10), octal (base 8) e hexadecimal (base 16).

Os inteiros decimais são os que usamos normalmente no dia-a-dia, dígitos de 0 a 9. Exemplo : `int tamanho = 343;`

Os inteiros octais usam somente dígitos de 0 a 7. Em java o inteiro octal é representado na forma octal com a inclusão de um zero na frente do número.

Exemplo: `int oito = 010; // Igual ao 8 decimal`

Os inteiros hexadecimais são construídos com o uso de 16 símbolos distintos. A representação hexadecimal utiliza os dígitos de 0 a 9 e A(10), B(11), C(12), D(13), E(14) e F(15). Em java a representação é feita incluindo o prefixo 0x.

Exemplo: `int quinze = 0xf; // Igual ao 15 em decimal`

Literais de ponto flutuante

Os números de ponto flutuante são definidos com um número com um símbolo decimal e outro número que representa a fração.

Exemplo: `double d = 12345.6789;`

Literais de booleanos

Literais booleanos só podem ser representados como true ou false.

Exemplo: `boolean isVerdadeiro = true;`

Literais de caracteres

O literal char é representado por um único caracter entre aspas simples. Os literais também podem ser representados com o valor Unicode, acrescentando o prefixo \u.

Exemplo:

```
char n = 'N'; //Letra N representada normalmente
char nUnicode = '\u004E'; //Letra N em formato Unicode
char nInteiro = 78; //Letra N em inteiro
```

Literais para strings

Um literal de string é a representação do código-fonte para o valor de um objeto String. São palavras escritas dentro do código.

Exemplo:

```
String curso = "Curso formação JAVA";
```

Conversão de tipos primitivos

As conversões podem ser *implícitas* ou *explícitas*. Uma conversão implícita significa que você não precisa escrever algum código, ela será automática. Normalmente, uma conversão implícita ocorre quando fazemos uma transformação que envolva ampliação, ou seja, quando tentamos inserir um container menor em um maior. O contrário disto é a conversão explícita, em que você estará informando ao compilador que conhece o perigo e aceita toda a responsabilidade.

Exemplo de conversão implícita:

```
int a = 100;
long b = a;
```

Exemplo de conversão explícita:

```
float a = 100.001f;
int b = (int)a; //vai perder a parte fracionária
```

Variáveis de instância primitivas

As variáveis de instância sempre são iniciadas com valor padrão quando o objeto é criado. A seguir veja uma tabela dos tipos e seus respectivos valores padrões:

Tipo de variável	Valor padrão
Variável de referência de objeto	Null
byte, short, int, long	0
float, double	0.0
Boolean	false
Char	'\u0000'

Passagem de variáveis para métodos

Quando for passada uma variável de objeto para um método, temos que lembrar que estaremos passando na verdade a referência ao objeto e não o próprio objeto. A variável de referência contém bits que representam uma maneira de acessar o objeto na memória heap. Na verdade a linguagem Java utiliza a semântica de passagem por cópia, tanto para primitivos quanto para objetos. No caso dos objetos, é passada a cópia da variável de referência. Já nos primitivos a passagem se dá por cópia do valor da variável.

Arrays

Em java, os arrays são objetos que armazenam diversas variáveis do mesmo tipo. Eles podem conter variáveis de referência ou primitivos.

Declarando arrays

Os arrays são declarados através da descrição do tipo de elemento que armazenarão, os quais podem ser um objeto ou um tipo primitivo, seguido de colchetes a direita ou esquerda do identificador.

Exemplos:

```
int array1 []; //unidimensional
int array2 [][]; //bidimensional
int array3 [][][]; //tridimensional
int [][] array4 []; //válido
```

Construindo arrays

Construir um array significa criar o objeto de array na pilha (onde todos os objetos residem), em outras palavras, usar a palavra-chave new com o tipo do array. Para criar um objeto de array, Java terá que saber quanto espaço alocar na pilha, portanto você precisa especificar o tamanho do array no momento da construção.

Exemplos:

```
int array1 [] = new int [5]; //unidimensional com 5 posições
int array2 [][] = new int [6][]; //obrigatório o tamanho da
primeira dimensão apenas
int array3 [][] = {{1,2}, {3,4}, {5,6}}; //array anônimo
int array4 [] = new int [3];
array4[0] = 1; //índice sempre iniciando em zero
array4[1] = 2;
array4[2] = 3;
```

Blocos de inicialização

Os blocos de inicialização são o terceiro lugar onde as operações podem ser realizadas, antes deles se encontram os métodos e construtores já mencionados. Existem dois tipos de blocos de inicialização, os estáticos e os de instância, esses blocos geralmente são utilizados para trechos de códigos que devam ser compartilhados entre os construtores.

Exemplo:

```
package com.flexxo;

public class BlocosInicializacao {

    static {
        System.out.println("Roda quando a classe é
carregada");
    }

    {
        System.out.println("Roda depois da chamada super do
construtor");
    }

    {
        System.out.println("Rodam na ordem em que
aparecem");
    }

    public static void main(String[] args) {
        BlocosInicializacao a = new BlocosInicializacao();
    }
}
```

Wrappers

As classe wrapper da API java servem para dois propósitos principais:

- Fornecer um mecanismo para “empacotar” valores primitivos em um objeto de modo que possam ser incluídos em atividades reservadas a objetos, como ser adicionados a Conjuntos ou retornados de um método em que o tipo de retorno seja um objeto;
- Fornecer um conjunto de funções utilitárias para tipos primitivos. A maioria destas funções estão relacionadas com várias conversões de tipos primitivos em objetos String e vice-versa, de tipos primitivos e objetos String para bases, binárias, octais e hexadecimais.

Tabela dos tipos primitivos X Classes wrappers e seus respectivos construtores

Tipo primitivo	Classe wrapper	Argumentos construtor
Boolean	Boolean	boolean ou string
Byte	Byte	byte ou string
Char	Character	char
Double	Double	double ou string
Float	Float	float ou string
Int	Integer	int ou string
Long	Long	long ou string
Short	Short	short ou string

Os métodos *valueOf*

Os dois métodos `valueOf()` fornecidos na maioria das classes wrappers, usam uma `String` para representar o tipo apropriado de seu primeiro argumento, sendo que o segundo método (quando fornecido) usa um argumento adicional, `int radix`, que indica em que base (binária, octal ou hexadecimal) o primeiro argumento foi representado.

Exemplo: `Integer i2 = Integer.valueOf("101011", 2);`

Os métodos *xxxValue()*

Quando você precisar converter o valor de um objeto wrapper numérico em um tipo primitivo, use um dos muitos métodos `xxxValue()`, cada uma das 6 classes wrappers numéricas possui 6 métodos, de modo que os objetos wrappers numéricos podem ser convertidos em qualquer tipo primitivo numérico.

Exemplo: `Integer i2 = Integer.valueOf(42);`
`byte byte1 = i2.byteValue();`

Os métodos *parseXxx()*

Este método tem a mesma função que os métodos `valueOf`, porém ao invés de retornar um objeto retorna um tipo primitivo.

Exemplo: `Integer i2 = Integer.parseInt("42");`

Autoboxing

Introduzido em Java 5, este recurso é conhecido por vários nomes: autoboxing, boxing (“encaixotar”) e unboxing (“desencaixotar”). O objetivo deste recurso é facilitar o processo de conversão entre classes wrappers e primitivos.

Anteriormente ao Java 5 para converter um `Integer` para um tipo primitivo `int`, era necessário o código abaixo.

```
Integer i = new Integer(42);  
int x = i.intValue();  
x++;
```

Agora o mesmo procedimento acima efetuado fica da seguinte forma (embora nos bastidores, no código compilado fique igual ao exemplo acima):

```
Integer i = new Integer(42);  
i++;
```

Boxing, ==, e equals

O objetivo do método equals, que todos os objetos wrappers possuem, é determinar se dois objetos são significativamente equivalentes, ou seja, se forem do mesmo tipo e tiverem o mesmo valor. Já o comparador == determina se duas instâncias de dois objetos são iguais, ou seja, se considerarmos que cada objeto tem um identificador único, o comparador == testa exatamente este identificador.

Porém, para economizar memória, duas instâncias dos seguintes objetos wrappers serão sempre iguais quando os seus valores primitivos forem os mesmos:

- Boolean
- Byte
- Character de \u0000 até \u007f
- Short e Integer de -128 e 127

Exemplo1:

```
Integer i = 10;
Integer i2 = 10;
if(i == i2) System.out.println("mesmo instancia de objeto");
if(i.equals(i2)) System.out.println("conteúdo igual");
```

Este exemplo produz a saída:

```
mesmo instancia de objeto
conteúdo igual
```

Exemplo2:

```
Integer i = 1000;
Integer i2 = 1000;
if(i != i2) System.out.println("instancia diferente");
if(i.equals(i2)) System.out.println("conteúdo igual");
```

Este exemplo produz a saída:

```
instancia diferente
conteúdo igual
```

Sobrecarga

Existem ainda 3 fatores que determinam a ordem de chamada dos métodos sobrecarregados de uma classe:

- Ampliação (converte para tipo de maior tamanho)
- Autoboxing (efetua a conversão de primitivo para objeto)
- Var-args (Quando o tipo de parâmetro é ...)

Quando uma classe tem métodos sobrecarregados, uma das tarefas do compilador é determinar qual método usar sempre que encontrar uma chamada ao método sobrecarregado.

Exemplo (com ampliação):

```
package com.flexxo;

public class Sobrerecarga {

    static void go(int x){System.out.println("int");}
    static void go(long x){System.out.println("long");}
    static void go(double x){System.out.println("double");}

    public static void main(String[] args) {
        byte b = 5;
        short s = 5;
        long l = 5;
        float f = 5.0f;
        go(b);
        go(s);
        go(l);
        go(f);
    }
}
```

Que produz a saída:

```
int
int
long
double
```

Exemplo (com boxing):

```
package com.flexxo;

public class Sobrerecarga {

    static void go(Integer x){System.out.println("Integer");}
    static void go(long x){System.out.println("long");}

    public static void main(String[] args) {
        int i = 5;
        go(i);
    }
}
```

Que produz a saída:

```
long
```

Desta forma podemos concluir que a ampliação é preferida ao boxing e também é preferida em vez do parâmetro var-args. Ou seja, o compilador sempre tentará ampliar o tipo da variável, caso nenhum método possua o parâmetro ampliado, irá buscar um método com os parâmetros boxing, caso não ache mesmo assim, irá procurar um método que corresponda com o tipo dos parâmetros var-args. Desta forma os códigos existentes anteriormente a versão 5 do Java continuaram funcionando com a versão java 5.

Regras da sobrecarga

Como podemos ver é possível combinar com sucesso var-args com ampliação ou com boxing, basta seguirmos as regras abaixo:

- A ampliação de primitivos usa o “menor” argumento possível do método;
- Usados individualmente, boxing e var-args são compatíveis com a sobrecarga;
- Não é possível ampliar de um tipo wrapper para outro;
- Não é possível ampliar e depois fazer boxing (Um int não pode se tornar um Long);
- É possível fazer boxing e depois ampliar (Um int pode se tornar um Object, via Integer)

Garbage collector

Visão geral do gerenciamento de memória e coletor de lixo

O coletor de lixo da linguagem Java fornece uma solução automática para o gerenciamento de memória. Na maioria dos casos ele o livrará de ter que adicionar alguma lógica de gerenciamento de memória em seu aplicativo. A desvantagem da coleta de lixo automática é que você não poderá controlar exatamente quando ela será ou não executada.

Visão geral do coletor de lixo em java

O principal objetivo do coletor de lixo é encontrar e excluir objetos que não possam ser alcançados. Ser alcançável significa que alguma Thread da aplicação possui uma referência para este objeto, caso contrário o objeto será considerado inalcançável e poderá ser excluído pelo GC.

Quando o coletor é executado ?

O coletor de lixo é controlado pela Virtual Machine. A JVM decide quando executá-lo. Dentro do programa Java você pode solicitar a JVM (System.gc ou Runtime.getRuntime().gc()) para executar o coletor de lixo, mas não haverá garantia alguma de que ele o fará.

Como o coletor de lixo funciona ?

Você não conseguirá ter certeza. Pode ouvir que o coletor de lixo usa um algoritmo de marcação e eliminação, e isso poderia ser verdade para alguma implementação JAVA específica, mas a especificação da linguagem não garante nenhuma implementação específica. O conceito importante a compreender é quando um objeto torna-se qualificado para coleta do GC.

Anulando a referência

A primeira maneira de remover a referência a um objeto é configurar como null a variável de referência que estiver apontando para ele.

Exemplo:

```
StringBuilder sb = new StringBuilder("Formação JAVA");  
sb = null;
```

Também podemos desassociar uma variável de referência de um objeto configurando-a para referenciar outro objeto.

Exemplo:

```
StringBuilder sb1 = new StringBuilder("Formação JAVA");  
StringBuilder sb2 = new StringBuilder("Programador");  
sb1 = sb2; //A partir daqui o objeto sb1  
           //está qualificado para coleta
```

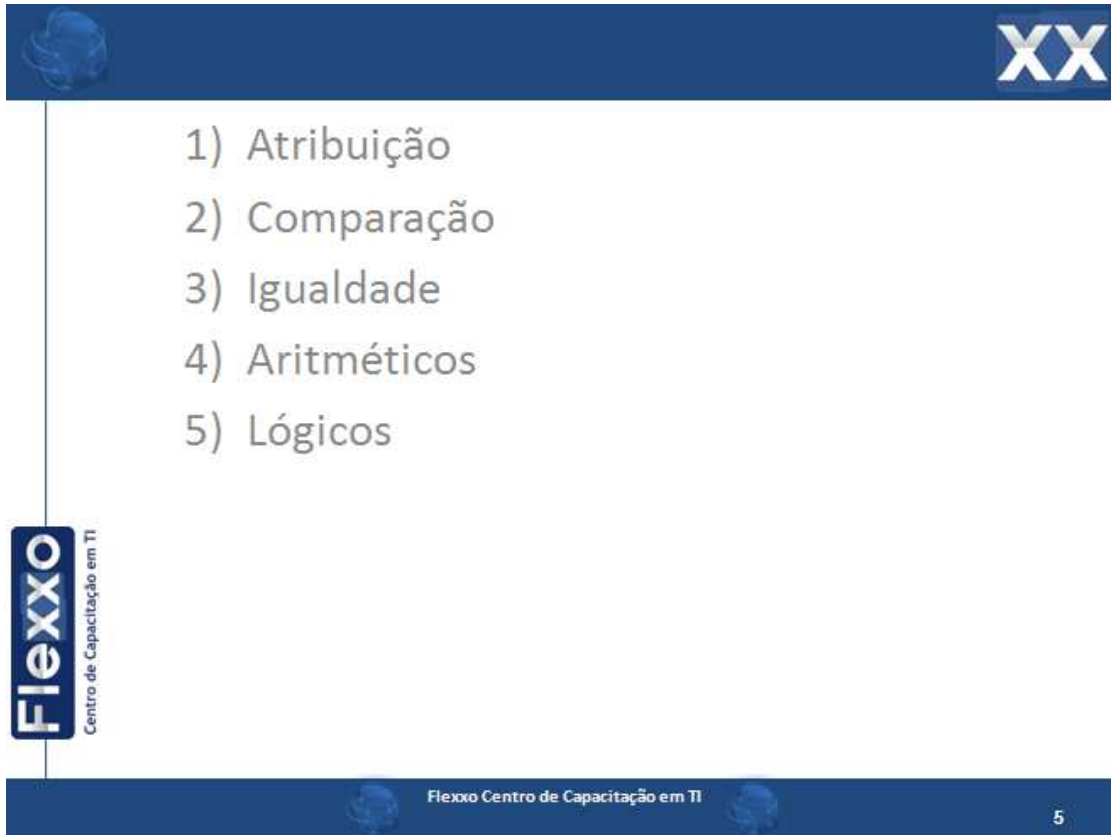
Os objetos criados em um método também são qualificados para coleta, ou seja, quando um método for chamado qualquer variável local que for criada só existirá durante a execução do método.

Método finalize

A linguagem Java fornece um mecanismo para que algum código seja executado imediatamente antes que um objeto seja excluído pelo coletor de lixo. Esse código ficará em um método chamado finalize() que todas as classes herdam da classe Object.

Centro de Capacitação em TI

Operadores



1) Atribuição
2) Comparação
3) Igualdade
4) Aritméticos
5) Lógicos

Flexxo
Centro de Capacitação em TI

Flexxo Centro de Capacitação em TI 5

Atribuição

Operadores de atribuição compostos

Na verdade, existem cerca de 11 operadores de atribuição compostos, mas apenas veremos os quatro mais utilizados +=, -=, *=, /=.

Exemplo sem utilizar operadores compostos:

```
int y = 10;  
int x = 10;  
y = y - 6;  
x = x + 2 * 5;  
System.out.println(y);  
System.out.println(x);
```

Exemplo utilizando operadores compostos:

```
int y = 10;  
int x = 10;  
y -= 6;  
x += 2 * 5;  
System.out.println(y);  
System.out.println(x);
```

Comparação

Operadores de comparação

Os operadores de comparação sempre resultam em um valor booleano (true ou false). Esse valor booleano é geralmente utilizado em testes if. Os operadores de comparação mais utilizados são: < (menor), <= (menor igual), > (maior), >= (maior igual).

Exemplo:

```
int y = 10;
int x = 5;
if(x > 10){

}
if(y < 6){

}
if(x >= 10){

}
if(y <= 6){

}
```

Igualdade

Operadores de igualdade

Os operadores de igualdade em Java, são == (igual) e != (diferente). Cada comparação pode envolver: dois números (incluindo o tipo char), dois valores booleanos ou duas variáveis de referência de objeto. Não é possível comparar tipos incompatíveis.

Igualdade de tipos primitivos

Para tipos primitivos, mesmo que os tipos sejam diferentes, se os valores forem iguais, o operador == retornará true.

Exemplo:

```
System.out.println("character 'a' == 'a' ? " + ('a' == 'a'));
System.out.println("character 'a' == 'b' ? " + ('a' == 'b'));
System.out.println(" 5 != 6 ? " + (5 != 6));
System.out.println(" 5.0 == 5L ? " + (5.0 == 5L));
System.out.println(" true == false ? " + (true == false));
```

A saída será:

```
character 'a' == 'a' ? true
character 'a' == 'b' ? false
5 != 6 ? true
5.0 == 5L ? true
true == false ? false
```

Igualdade de variáveis de referência

O exemplo abaixo mostra que as variáveis a e c estão se referenciando a mesma instância de um objeto JButton. Neste caso o operador == não testará se dois objetos são “significativamente equivalentes”.

Exemplo:

```
JButton a = new JButton("Exit");
JButton b = new JButton("Exit");
JButton c = b;

System.out.println(" a == b ? " + (a == b));
System.out.println(" b == c ? " + (b == c));
```

A saída será:

```
a == b ? false
b == c ? true
```

Igualdade para enums

Para enums é possível utilizar tanto o == quanto o equals, pois enums não podem ser criadas novas instâncias.

Exemplo:

```
package com.flexxo;

public class EnumEqual {

    enum Color {RED, BLUE}

    public static void main(String[] args) {
        Color c1 = Color.RED;
        Color c2 = Color.RED;
        if(c1 == c2){System.out.println(" == ");}
        if(c1.equals(c2)){System.out.println(" equals ");}
    }
}
```

Operador instanceof

O operador instanceof é usado somente com variáveis de referência de objeto, e você pode empregá-lo para verificar se um objeto é de um tipo específico.

Exemplo:

```
package com.flexxo;

public class InstanceOf {

    public static void main(String[] args) {
        String s = new String("foo");
        if(s instanceof String){
            System.out.println("s é uma String");
        }
    }
}
```

A saída é:

s é uma String

Aritméticos

Os operadores aritméticos são + (soma), - (subtração), * (multiplicação), / (divisão), % (resto).

Acréscimo e decréscimo

Em java temos dois operadores que incrementam ou decrementam a variável em exatamente 1 unidade. Esses operadores são compostos por 2 sinais de adição (++) ou dois sinais de subtração (--). Esses operadores podem ser de duas formas:

- Pré – fixado: inserido antes da variável e executa antes do comando da linha.

Exemplo:

```
int x = 10;
System.out.println(x++);
System.out.println(x);
```

E resultado fica:

10
11

- Pós – fixado: inserido depois da variável e executa depois do comando da linha.

Exemplo:

```
int x = 10;
System.out.println(++x);
System.out.println(x);
```

E o resultado:

11

11

If ternário

Este comando tem o mesmo objetivo do if, porém executa em apenas uma linha. Sintaxe: `x = (expressão booleana) ? valor a retornar se true : valor a retornar se false.`

Exemplo:

```
String a = (3 > 4 ? "3 > 4" : "3 < 4" );
```

Lógicos

Operadores lógicos de abreviação

Os operadores lógicos de abreviação mais utilizados são `&&` (e) e `||` (ou). O recurso de abreviação do operador `&&` consiste no fato “de ele não perder tempo em avaliações inúteis”. O operador de abreviação `&&` avaliará o lado esquerdo da operação primeiro e se esse operando tiver um resultado false, ele não examinará o lado direito da equação. Já para o operador `||`, caso o primeiro operando seja true ele não fará avaliar o lado direito.

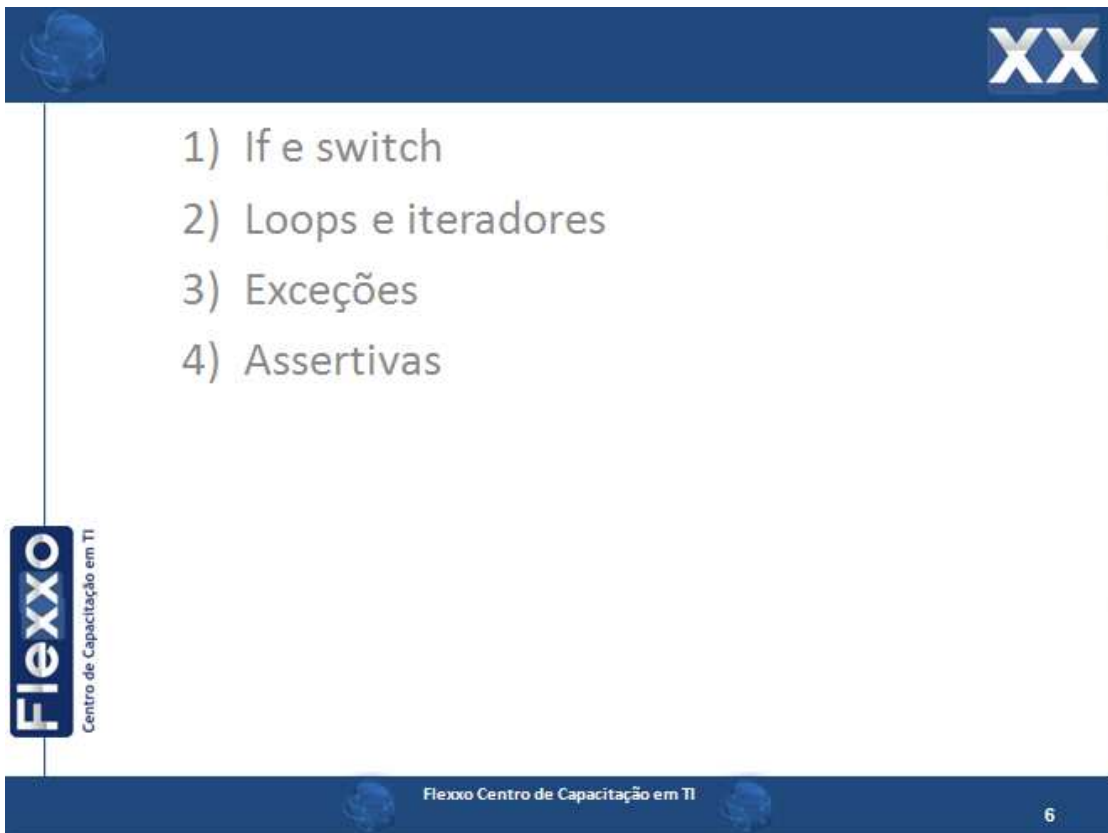
Operadores lógicos de não-abreviação

Os operadores lógicos de não abreviação mais utilizados são `&` (e) e `|` (ou). E eles sempre avaliam todos os lados da equação, independente do primeiro operando.

Operadores lógicos ^ e !

Existem ainda os operadores lógicos `^` (ou exclusivo - XOR) e `!` (negação). Para uma operação de OU exclusivo ser verdadeira, exatamente 1 operando deve ser verdadeiro. Desta forma este operador se relaciona com os operadores não-abreviados, visto que terá que avaliar todos os operandos da expressão.

Controle de fluxo, exceções e assertivas



1) If e switch
2) Loops e iteradores
3) Exceções
4) Assertivas

Flexxo
Centro de Capacitação em TI

Flexxo Centro de Capacitação em TI 6

if e switch

As instruções if e switch, geralmente são conhecidas como instruções de decisão. Quando você usar instruções de decisão em seu programa, estará solicitando que ele avalie uma expressão específica a fim de determinar para que parte do código deve seguir a execução.

if-else

Sintaxe do if básico :

```
if (expressao booleana) {  
    System.out.println("expressao booleana true");  
}
```

A expressão entre parêntese deve ter como resultado um valor booleano true ou false. Normalmente você testará algo para saber se é verdadeiro e, em seguida, executará um bloco de código se o resultado for mesmo verdadeiro e opcionalmente outro bloco de código se não o for.

Exemplo:

```
int x = 10;
if (x > 11){
    System.out.println("expressao booleana true");
}else{
    System.out.println("expressao booleana false");
}
```

As instruções if else podem também não possuírem chaves, isto é possível apenas quando for apenas uma instrução dentro do if ou else, porém sempre é recomendado que seja colocado as chaves por uma questão de legibilidade de código.

Exemplo:

```
int x = 10;
if (x > 11)
    System.out.println("expressao booleana true");
else
    System.out.println("expressao booleana false");
```

Além disso é possível aninhar instruções, quando a necessidade é ter várias condições.

Exemplo não aninhando o if else:

```
int x = 10;
if (x > 11){
    System.out.println("x > 11");
}else{
    if(x > 10){
        System.out.println("x > 10");
    }else{
        if(x > 9){
            System.out.println("x > 9");
        }else{
            System.out.println("x <= 9");
        }
    }
}
```

Exemplo aninhando if else:

```
int x = 10;
if (x > 11) {
    System.out.println("x > 11");
} else if (x > 10) {
    System.out.println("x > 10");
} else if (x > 9) {
    System.out.println("x > 9");
} else {
    System.out.println("x <= 9");
}
```

switch

Switch é um comando que auxilia quando temos a necessidade de executar vários testes if em sequencia.

Sintaxe:

```
switch (variável) {  
    case constante 1: bloco de código  
    case constante 2: bloco de código  
    default: bloco de código  
}
```

Diferente do if, o switch só testa igualdades e só pode avaliar com tipos char, byte, short, int e enum. A variável que o switch recebe por parâmetro é o valor que ele irá utilizar para verificar qual dos “cases” é igual, neste caso o bloco de código deste case será executado. Caso nenhum dos cases seja igual ao valor da variável, o fluxo do código será encaminhado para o bloco de código da instrução default caso esta exista. Porém a execução do switch tem uma particularidade, que acontece quando o fluxo de execução chega ao bloco de código de um determinado case. Se não colocarmos uma instrução break no final do bloco de código, a execução continuará no case abaixo.

Exemplo (sem breaks) :

```
int x = 10;  
switch (x) {  
    case 10: System.out.println("10");  
    case 11: System.out.println("11");  
    default: System.out.println("default");  
}
```

A saída será:

```
10  
11  
default
```

Exemplo (com breaks) :

```
int x = 10;  
switch (x) {  
    case 10: System.out.println("10"); break;  
    case 11: System.out.println("11"); break;  
    default: System.out.println("default"); break;  
}
```

A saída será:

```
10
```

Loops e iteradores

Os loops da linguagem Java vêm em três versões: while, do e for (a partir da versão 5 o loop for tem duas variantes). Todas as três permitem que você repita a execução de um bloco de código até que determinada condição seja satisfeita ou durante uma quantidade específica de iterações.

while

Este loop será adequado em cenários nos quais não se sabe quantas vezes o bloco ou instrução terá que ser executado, porém sabe-se que este deve executar até que uma condição seja verdadeira.

Exemplo:

```
int x = 10;
while (x < 10){
    System.out.println("x = " + x++);
}
```

do while

O loop do while é semelhante ao while exceto pelo fato de que a expressão não é avaliada até que o código tenha sido executado. Desta forma é garantido que pelo menos uma vez será executado o bloco de código.

Exemplo:

```
int i = 0;
do{
    System.out.println("i = " + i);
}while(i > 0);
```

for

O loop for será útil para o controle de fluxo quando soubermos a quantidade de vezes que o bloco de código deverá executar. A declaração deste loop possui três partes:

- *Declaração e inicialização de variáveis;*
- *A expressão booleana;*
- *A expressão de iteração;*

Sintaxe:

```
for(/*inicialização*/; /*Condição*/; /*Iteração*/){
    /*Corpo do loop*/
}
```

Exemplo:

```
for(int i = 0; i < 10; i++){
    System.out.println(i);
}
```

A primeira parte da instrução for permite que você declare e inicialize nenhuma, uma ou diversas variáveis do mesmo tipo. Em caso de declarar mais que uma, estas deverão ser separadas por vírgula. A declaração e inicialização ocorrem antes de qualquer outra coisa em um loop for e apenas uma vez.

Exemplo:

```
for(int i = 0, y = 0; i < 10; i++){
    System.out.println(i);
}
```

A segunda parte é a expressão booleana, ou seja, o for somente executará até que a variável seja verdadeira. O teste condicional sempre é executado antes do corpo do for ser executado.

A terceira parte é a expressão de iteração. Esta parte sempre é executada depois de cada execução do bloco de código.

Podemos também finalizar a execução do for antes do término das iterações através dos seguintes comandos: `break`, `return` ou `System.exit()`.

Além disso, nenhuma das três partes do for são obrigatórias, desta forma o exemplo abaixo é perfeitamente válido, porém executará eternamente.

Exemplo:

```
for( ; ; ){  
    System.out.println("Loop eterno");  
}
```

for aprimorado

Introduzido na versão 5 do Java, simplifica a tarefa de fazer um loop através de um array ou conjunto.

Sintaxe:

```
for (declaração : expressao) {  
    //corpo  
}
```

A primeira parte do for deve ser a declaração de uma variável de tipo compatível com o dos elementos do array. Esta variável só estará disponível dentro do loop e conterá o valor do elemento atual do array.

A segunda parte é o array ou conjunto a iterar. Pode ser também a chamada de um método que retorne um destes dois tipos de objetos.

Exemplo usando for normal:

```
int array[] = { 1, 2, 3, 4, 5 };  
for (int i = 0; i < array.length; i++) {  
    System.out.println(" posicao " + i + " = " + array[i]);  
}
```

Exemplo usando for aprimorado:

```
int array[] = { 1, 2, 3, 4, 5 };  
for (int posicao : array) {  
    System.out.println(" posicao = " + posicao);  
}
```

break e continue

Os comandos `break` e `continue` são utilizados para encerrar a execução de um loop inteiro (`break`) ou apenas da iteração atual (`continue`).

Exemplo (`break`):

```
int array[] = { 1, 2, 3, 4, 5 };
for (int posicao : array) {
    if(posicao == 3){
        break;
    }
    System.out.println(" posicao = " + posicao);
}
```

A saída será:

```
posicao = 1
posicao = 2
```

Exemplo (continue):

```
int array[] = { 1, 2, 3, 4, 5 };
for (int posicao : array) {
    if(posicao == 3){
        continue;
    }
    System.out.println(" posicao = " + posicao);
}
```

A saída será:

```
posicao = 1
posicao = 2
posicao = 4
posicao = 5
```

Exceções

A manipulação de exceções permite que os desenvolvedores detectem erros facilmente sem escrever um código especial para testar valores retornados. Melhor ainda, nos permite manter um código de manipulação de exceções nitidamente separado do código que gerará a exceção. Além disso, permite que o mesmo código de manipulação de exceções lide com as diferentes exceções possíveis.

try catch

O termo exceção significa “condição excepcional” e é uma ocorrência que altera o fluxo normal do programa. Várias coisas podem levar a exceções, incluindo falhas no hardware, exaustão de recursos e erros. Quando um evento excepcional ocorre Java diz-se que uma exceção será lançada. O trecho de código que é responsável por fazer algo com a exceção é chamado de manipulador de exceções. Portanto, precisamos de uma maneira para informar à JVM que código executar quando determinada exceção ocorrer. Para isso usamos as palavras try e catch.

Exemplo:

```
package com.flexxo;

public class ExemploTryCatch {

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

```

        ExemploTryCatch a = null;
        //Vai lançar um nullpointer
        a.getValor();
        System.out.println("try");
    } catch (RuntimeException e) {
        System.out.println("Catch");
    }
}
private String getValor(){
    return " abc ";
}
}

```

A saída do código acima é:

Catch

finally

Utilizado para liberação de recursos, limpeza de variáveis ou outras coisas que necessitem ser executadas após a execução do bloco try ou catch, ou seja, sempre será executado independente de uma exceção ser lançada ou não.

Exemplo:

```

package com.flexxo;

public class ExemploTryCatch {

    public static void main(String[] args) {
        try{
            ExemploTryCatch a = null;
            //Vai lançar um nullpointer
            a.getValor();
            System.out.println("try");
        } catch (RuntimeException e) {
            System.out.println("Catch");
        } finally{
            System.out.println("finally");
        }
    }
    private String getValor(){
        return " abc ";
    }
}

```

A saída será:

Catch
finally

Propagando exceções

Quando um método não fornece uma cláusula catch para uma exceção específica, diz-se que esse método está propagando a exceção para o método chamador e assim por diante até que um dos métodos chamadores esteja tratando a

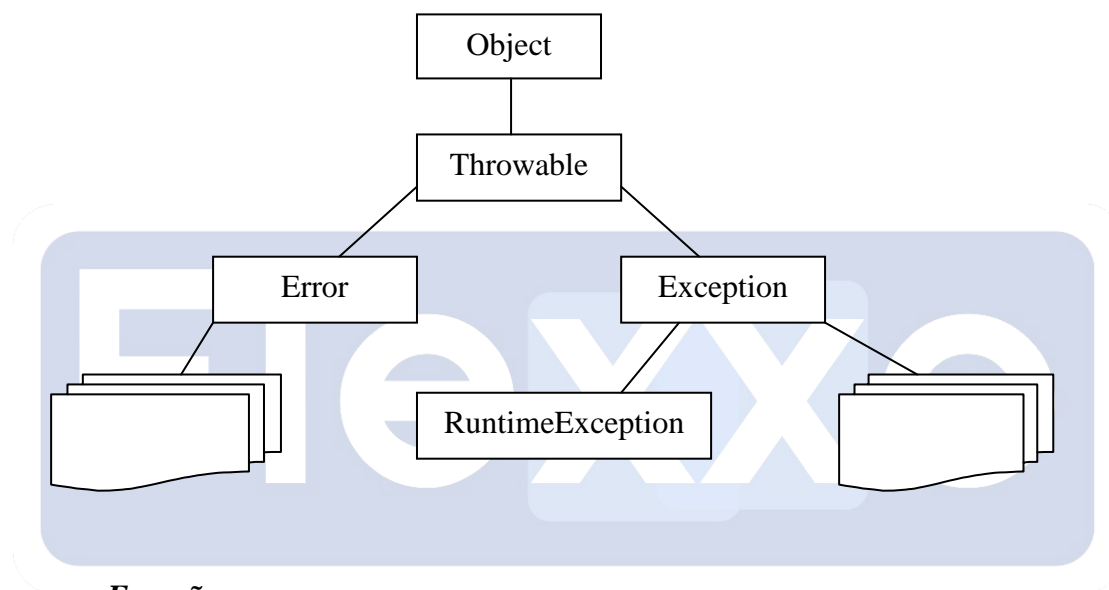
exceção lançada, caso esse erro não seja tratado por nenhum método chamador, o programa acabará por finalizar a execução.

Hierarquia de exceções

Existem dois tipos de exceções:

- Verificadas
- Não verificadas

As verificadas são todas as exceções que estendem a classe Exception, essas obrigatoriamente devem ser tratadas, caso contrário teremos erro de compilação. Já as não verificadas são todas as exceptions que estendem a classe RuntimeException e estas não precisam ser tratadas.



Exceções comuns

Abaixo uma tabela contendo as exceções mais comuns e seus respectivos motivos:

Exceção	Motivo
ArrayIndexOutOfBoundsException	Lançada ao tentar acessar um array com um valor de índice inválido (negativo ou maior que a extensão do array)
ClassCastException	Lançada ao tentar converter uma variável de referência em um tipo que não passa no teste instanceof.
IllegalArgumentException	Lançada quando um método recebe um argumento formatado de forma diferente do que o método espera.
IllegalStateException	Lançada quando o estado do ambiente não bate com a operação que está sendo feita. Por exemplo : usando-se um Scanner que já foi fechado.
NullPointerException	Lançada ao tentar usar um método ou

	variável de classe cuja a referencia da variável seja null.
NumberFormatException	Lançada ao tentar converter uma string em número e esta string não representar um número.
AssertionError	Lançada quando o teste booleano de uma instrução retorna false
ExceptionInInitializerError	Lançada ao tentar inicializar uma variável estática de um bloco de inicialização.
StackOverflowError	Lançada quando um método faz recursões muito profundas.
NoClassDefFoundError	Lançada quando a JVM não consegue encontrar uma classe de que precisa.

Assertivas

Adicionadas à linguagem Java na versão 1.4, as assertivas permitem que você teste suas suposições durante o desenvolvimento, sem o desgaste de escrever manipuladores para exceções que se supõe que nunca ocorrerão, uma vez que o programa tiver saído da fase de desenvolvimento e for totalmente distribuído.

Vamos simular o uso de uma assertiva, imaginando que um número passado para um método nunca possa ser negativo.

Exemplo (sem uso de assertiva):

```
public void metodo(int a){
    if (a > 0){
        // faz algo
    }else{
        System.out.println("Número negativo não pode!");
    }
}
```

Exemplo com uso de assertivas:

```
public void metodo(int a){
    //Se a >= 0 não acontece nada
    //Se for menor é lançado uma AssertionError
    assert(a >= 0) : "Número negativo não pode!";
    // do code here
}
```

Porém no momento da execução poderemos optar por ativar ou não as assertivas, por padrão elas são desabilitadas.

Para ativá-las utilize a opção `-ea` na linha de comando e para desabilitar `-da`.

Exemplo (rodando no prompt):

```
java -ea TesteAssertivas
```

String, E/S, formatação e parsing



- 1) String, StringBuilder e StringBuffer
- 2) Manipulação de arquivos
- 3) Serialização
- 4) Data, número e moeda
- 5) Parsing, tokenização e formatação

String, StringBuilder e StringBuffer

String

O principal conceito a se entender é que, uma vez criado um objeto String, ele nunca poderá ser modificado. A manipulação de strings é um aspecto fundamental da maioria das linguagens de programação, em Java cada caractere de uma string é um caractere Unicode de 16 bits.

Assim, um dos objetivos principais de qualquer linguagem de programação é fazer uso eficiente de memória. Desta forma é muito comum que strings literais ocupem grandes espaços da memória, e que geralmente haja muita redundância dentro do universo de strings literais de um programa. Para tornar a linguagem mais eficiente no uso da memória, a JVM deixa reservada uma área especial chamada “pool de constante de strings”. Quando o compilador encontra uma string literal, verifica o pool para ver se já existe uma idêntica, caso exista atribui esta referência a variável ao invés de criar uma nova string.

Daí o motivo das Strings serem inalteráveis, se diversas variáveis de referência apontam para a mesma String (que estava no pool), imagine o caos que seria se alguém alterasse o conteúdo da String, desta forma todos os programas que utilizassem aquela String do pool teriam seu valor alterado “do nada” durante a execução de um programa qualquer.

Desta forma sempre que fizermos alguma operação que altere o conteúdo da String (concatenação, alterar para maiúsculo... etc), o compilador criará um novo objeto e atribuirá a variável de referência que estiver recebendo.

Exemplo:

```
//caso esta literal exista no pool
//não irá criar o objeto novamente
String x = "Java";
//idem para " abc "
//porém será criado um novo objeto
//contendo o conteúdo de x e " abc ".
//A referência deste novo objeto será
//atribuído a variável x
x = x.concat(" abc ");
System.out.println(x);
```

Métodos importantes

Abaixo uma tabela contendo algum dos métodos mais importantes da classe String:

<i>Método</i>	<i>O que ele faz ?</i>
charAt(int index)	Retorna o caracter localizado no índice especificado
concat(String s)	Anexa uma String ao final de outra, (“+” faz a mesma coisa que este método.
equalsIgnoreCase(String s)	Determina a igualdade de duas Strings independente da caixa.
length()	Retorna o número de caracteres de uma string.
replace(char old, char new)	Substitui as ocorrências de um caractere por outro caractere
substring(int begin, int end)	Retorna uma parte de uma string
toLowerCase()	Retorna uma string com os caracteres maiúsculos convertidos para minúsculo
toUpperCase()	Retorna uma string com os caracteres minúsculo convertidos para maiúsculos
trim()	Remove os espaços em branco no início e no fim de Strings

StringBuffer e StringBuilder

As classes StringBuffer e StringBuilder devem ser usadas quando tivermos a necessidade de efetuar muitas alterações em uma String. Como as Strings são imutáveis, logicamente estaremos criando muitos objetos na memória desnecessariamente. Objetos do tipo StringBuffer ou StringBuilder podem ser alterados repetidamente sem que para cada alteração seja criado um novo objeto.

A classe StringBuilder foi adicionada na versão 5 do Java. Ela tem exatamente os mesmo métodos que a StringBuffer, porém StringBuffer é synchronized, ou seja, é utilizada quando trabalhamos com Threads.

Exemplo:

```
StringBuilder s = new StringBuilder();  
s.append("Formação ");  
s.append("Java");  
System.out.println(s);
```

A saída será:

Formação Java

Métodos importantes

A seguir uma tabela contendo os principais métodos destas classes:

<i>Método</i>	<i>O que ele faz ?</i>
append(String s)	Este método concatena a variável passada por parâmetro com o conteúdo atual da variável.
delete (int start, int end)	Este método remove a partir do índice start (iniciando em 0) até o índice end (iniciando em 1)
insert(int offset, String s)	Este método insere o conteúdo passado no segundo parâmetro na posição recebida no primeiro parâmetro.
reverse()	Inverte o conteúdo, ou seja, o primeiro caracter passa a ser o último, o segundo o penúltimo e assim sucessivamente.
toString()	Retorna o conteúdo em String

Manipulação de arquivos

Abaixo, um resumo das classes de E/S que você precisará saber para manipular arquivos em Java:

- *File*: A api diz que a classe File é uma “representação abstrata de nomes de caminho de arquivos e diretórios”. A classe File não é usada para ler ou escrever dados, é usada para trabalhar em um nível mais alto, criar arquivos vazios, procurar arquivos, apagar arquivos, criar diretórios e trabalhar com caminhos.

Exemplo:

```
package com.flexxo;  
import java.io.File;  
import java.io.IOException;  
  
public class ExemploFile {  
  
    public static void main(String[] args) {  
        try {
```

```
        boolean newFile = false;
        File file = new
File("C:\\Users\\darmino\\Documents\\testeFile.txt");
        // Verifica se existe o arquivo
        System.out.println(file.exists());
        // Cria um novo arquivo caso não exista
        // retorna true se criou
        newFile = file.createNewFile();
        System.out.println(newFile);
        System.out.println(file.exists());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

A na primeira execução será:

```
false
true
true
```

- **FileReader**: Esta classe é usada para ler arquivos de caracteres. Os seus métodos `read()` são de nível relativamente baixo, permitindo que você leia caracteres isolados, todo o stream de caracteres ou um número fixo de caracteres. Os `FileReaders` normalmente são encapsulados por objetos de nível mais alto, como `BufferedReaders`, os quais melhoram o desempenho e fornecem meios ainda mais convenientes.
- **BufferedReader**: Esta classe é usada para tornar classes `Reader` de nível mais baixo, como `FileReader`, mais eficientes e fáceis de usar. Comparado com `FileReaders`, `BufferedReaders` lêem pedaços relativamente grandes de dados retirados do arquivo de uma só vez, e mantém esses dados no buffer. Quando solicitamos um caractere ou a linha seguinte este é buscado do buffer o que reduz o tempo de acesso ao disco. Além disso, esta classe fornece métodos como `readLine()` que lhe permite obter a próxima linha de caracteres do arquivo.
- **FileWriter**: Essa classe é usada para escrever em arquivos de caracteres. Os seus métodos `write()` lhe permitem escrever caracteres ou `Strings` em um arquivo. Os `FileWrites` geralmente são encapsulados por objetos `Writer` de nível mais alto, tais como `BufferedWriters` ou `PrintWriters`, que fornecem melhor desempenho e métodos de alto-nível mais flexíveis para escrever dados.
- **BufferedWriter**: Esta classe é usada para tornar classes de baixo nível, como `FileWriters`, mais eficientes e fáceis de usar. Comparados com `FileWriters`, `BufferedWriters` escrevem pedaços relativamente grandes de dados no arquivo de uma só vez, minimizando o número de vezes as demoradas operações de escrita de arquivos. Além disso, esta classe fornece o método `newLine()` que facilita a criação automática de separadores de linhas específicos da plataforma.

- *PrintWriter*: Esta classe foi melhorada significativamente na versão 5. Devido aos novos métodos e construtores, ficou possível utilizar *PrintWriters* em lugares onde anteriormente, precisava que uma classe *Writer* fosse encapsulado com um *FileWriter* e/ou *BufferedWriter*.

Combinando classes de E/S

Todo o sistema de E/S em Java foi elaborado tendo-se em mente a idéia de se usar diversas classes combinadas. Essa combinação de classes de E/S é as vezes chamada de wrapping e as vezes de encadeamento. Cada classe do pacote *java.io* tem um propósito bem específico e estas classes foram elaboradas para serem combinadas entre si de inúmeras formas, para lidar com uma ampla gama de situações. Na tabela abaixo será listada as principais classes do pacote *java.io* e suas respectivas interdependências. Assim quando formos programar ficará fácil sua utilização:

<i>Classe</i>	<i>Estende de</i>	<i>Argumento do/s construtor/es</i>	<i>Métodos principais</i>
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File String OutputStream Writer	close() flush() format(), printf() print(), println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read(), readLine()

Exemplo (criando e lendo um arquivo com FileWriter e FileReader):

```
package com.flexxo;

import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ExemploFileReaderFileWriter {
    public static void main(String[] args) {
        char [] in = new char [21];
        int size = 0;
        try{
            //Cria uma instancia do file indicando o arquivo
            // porém ainda não criou o arquivo.
            File file = new
File("C:\\Users\\darmino\\Documents\\testeFileWriter.txt");
            //Cria o arquivo
            FileWriter fw = new FileWriter(file);
            //Escreve no buffer
            fw.write("Escrevendo no arquivo");
            //Descarrega o buffer no arquivo
            fw.flush();
            //fecha o arquivo
            fw.close();
            //Cria um objeto FileReader
            FileReader fr = new FileReader(file);
            //lê o arquivo inteiro e joga na variável in
            //retorna a quantidade bytes lidos
            size = fr.read(in);
            //escreve o conteúdo do arquivo no console
            for(char c : in){
                System.out.print(c);
            }
            //fecha o arquivo
            fr.close();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Exemplo (criando e lendo um arquivo com BufferedWriter e BufferedReader):

```
package com.flexxo;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ExemploBufferedReaderBufferedWriter {
    public static void main(String[] args) {
        FileWriter fw = null;
        BufferedWriter bw = null;
        FileReader fr = null;
        BufferedReader br = null;
```

```

    try{
        //Cria uma instancia do file indicando o arquivo
        // porém ainda não criou o arquivo.
        File file = new
File("C:\\Users\\darmino\\Documents\\testeBufferedWriter.txt");
        //Cria o arquivo
        fw = new FileWriter(file);
        bw = new BufferedWriter(fw);
        //Escreve no buffer
        bw.write("Escrevendo no arquivo");
        //Descarrega o buffer no arquivo
        bw.flush();
        //Cria um objeto FileReader
        fr = new FileReader(file);
        br = new BufferedReader(fr);
        String linha = null;
        //enquanto o método readLine não retornar null
        //significa que existe linhas no arquivo
        while((linha = br.readLine()) != null){
            System.out.println(linha);
        }
    }catch(IOException e){
        e.printStackTrace();
    }finally{
        //fecha o arquivo
        if(bw != null){
            try {
                bw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(fw != null){
            try {
                fw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(br != null){
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(fr != null){
            try {
                fr.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
}

```

Exemplo (criando um arquivo com PrintWriter):

```
package com.flexxo;
```



```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class ExemploPrintWriter {
    public static void main(String[] args) {
        PrintWriter pw = null;
        try {
            pw = new
PrintWriter("C:\\Users\\darmino\\Documents\\testePrintWriter.tx
t");
            pw.println("Escrevendo no arquivo ");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            pw.flush();
            pw.close();
        }
    }
}
```

Serialização

Imagine que você queira salvar o estado de um ou mais objetos, se Java não tivesse a serialização, você teria de usar uma das classes de E/S para escrever os estado das variáveis de instância de todos os objetos que quisesse salvar. A pior parte seria tentar reconstruir novos objetos que fosse praticamente idênticos aos objetos que você estava tentando salvar. Você precisaria ter o seu próprio protocolo para maneira pela qual escreveria e restauraria o estado de cada objeto, caso contrário poderia acabar definindo variáveis com os valores errados.

ObjectOutputStream e ObjectInputStream

A serialização básica acontece com apenas dois métodos: um para serializar os objetos e escrevê-los em um stream (`ObjectOutputStream.writeObject()`) e outro para ler o stream e deserializar (`ObjectInputStream.readObject()`).

As classes `java.io.ObjectOutputStream` e `java.io.ObjectInputStream`, são consideradas como de alto nível, desta forma você as utilizará para encapsular classes de níveis mais baixo, tais como: `java.io.FileOutputStream` e `java.io.FileInputStream`.

Exemplo:

```
package com.flexxo;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Pessoa implements Serializable{
    private String desPessoa;

    public String getDesPessoa() {
        return desPessoa;
    }

    public void setDesPessoa(String desPessoa) {
        this.desPessoa = desPessoa;
    }
}

public class ExemploObjectOutputStreamEObjectInputStream {

    public static void main(String[] args) {
        Pessoa c = new Pessoa();
        c.setDesPessoa("Daniel Bruno Armino");
        try{
            FileOutputStream fs = new
FileOutputStream("C:\\Users\\darmino\\Documents\\testeSerializacao.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(c);
            os.close();

        }catch(Exception e){
            e.printStackTrace();
        }
        try{
            FileInputStream fs = new
FileInputStream("C:\\Users\\darmino\\Documents\\testeSerializacao.ser");
            ObjectInputStream os = new ObjectInputStream(fs);
            Pessoa ps = (Pessoa)os.readObject();
            os.close();
            System.out.println(ps.getDesPessoa());

        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Grafos de objetos

Quando serializamos um objeto, gravamos ele e todos as variáveis de instância. O problema ocorre quando uma destas variáveis for um objeto e este não implementar o tipo Serializable, neste caso quando salvarmos o objetos teremos o erro

java.io.NotSerializableException em tempo de execução. Podemos proceder de duas formas para resolver este problema, a primeira seria alterar os objetos que não implementam a interface Serializable. Caso isso não seja possível, devemos colocar o modificador transient na variável que referencia o objeto, assim este atributo não será salvo ao serializarmos o objeto.

Exemplo:

```
package com.flexxo;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Pessoa implements Serializable{

    private String desPessoa;
    private transient Endereco endereco;

    public String getDesPessoa() {
        return desPessoa;
    }

    public void setDesPessoa(String desPessoa) {
        this.desPessoa = desPessoa;
    }

    public Endereco getEndereco() {
        return endereco;
    }

    public void setEndereco(Endereco endereco) {
        this.endereco = endereco;
    }
}

class Endereco {
    private String desEndereco;

    public String getDesEndereco() {
        return desEndereco;
    }

    public void setDesEndereco(String desEndereco) {
        this.desEndereco = desEndereco;
    }
}

public class ExemploTransient {

    public static void main(String[] args) {
        Pessoa c = new Pessoa();
        c.setDesPessoa("Daniel Bruno Armino");
        Endereco endereco = new Endereco();
        endereco.setDesEndereco(" Avenida Rio Branco ");
        c.setEndereco(endereco);
        try{
```

```

        FileOutputStream fs = new
FileOutputStream("C:\\Users\\darmino\\Documents\\testeSerTrans.ser");
        ObjectOutputStream os = new ObjectOutputStream(fs);
        os.writeObject(c);
        os.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
    try {
        FileInputStream fs = new
FileInputStream("C:\\Users\\darmino\\Documents\\testeSerTrans.ser");
        ObjectInputStream os = new ObjectInputStream(fs);
        Pessoa ps = (Pessoa)os.readObject();
        os.close();
        //retorna null pois o atributo é transient
        System.out.println(ps.getEndereco());

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

Métodos writeObject e readObject

A serialização Java possui um mecanismo especial para quando necessitamos interferir no processo de serialização ou deserialização, por exemplo, acrescentando atributos no stream. Um conjunto de métodos privados que pode ser implementado na classe que, se presente será chamado automaticamente durante a serialização ou deserialização. É quase como se os métodos fossem definidos na interface Serializable, exceto pelo fato de que não são. Caso tivermos os métodos abaixo implementados na classe estes serão chamados:

- **private void writeObject(ObjectOutputStream os)**
- **private void readObject(ObjectInputStream is)**

Exemplo:

```

package com.flexxo;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Pessoa implements Serializable {

    private String desPessoa;
    private transient Endereco endereco;

    public String getDesPessoa() {
        return desPessoa;
    }
}

```

```

    public void setDesPessoa(String desPessoa) {
        this.desPessoa = desPessoa;
    }

    public Endereco getEndereco() {
        return endereco;
    }

    public void setEndereco(Endereco endereco) {
        this.endereco = endereco;
    }

    private void writeObject(ObjectOutputStream os) {
        try {
            os.defaultWriteObject();

            os.writeObject(this.getEndereco().getDesEndereco());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void readObject(ObjectInputStream is) {
        try {
            is.defaultReadObject();
            String desEndereco = (String) is.readObject();
            Endereco e = new Endereco();
            e.setDesEndereco(desEndereco);
            this.setEndereco(e);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}

class Endereco {
    private String desEndereco;

    public String getDesEndereco() {
        return desEndereco;
    }

    public void setDesEndereco(String desEndereco) {
        this.desEndereco = desEndereco;
    }
}

public class ExemploWriteObjectReadObject {

    public static void main(String[] args) {
        Pessoa c = new Pessoa();
        c.setDesPessoa("Daniel Bruno Armino");
        Endereco endereco = new Endereco();
        endereco.setDesEndereco(" Avenida Rio Branco ");
        c.setEndereco(endereco);
        try {
            FileOutputStream fs = new FileOutputStream(

```

```

"C:\\Users\\darmino\\Documents\\testeSerWriteRead.ser");
    ObjectOutputStream os = new ObjectOutputStream(fs);
    os.writeObject(c);
    os.close();

} catch (Exception e) {
    e.printStackTrace();
}
try {
    FileInputStream fs = new FileInputStream(
        "C:\\Users\\darmino\\Documents\\testeSerWriteRead.ser");
    ObjectInputStream os = new ObjectInputStream(fs);
    Pessoa ps = (Pessoa) os.readObject();
    os.close();
    //retorna null pois o atributo é transient

    System.out.println(ps.getEndereco().getDesEndereco());

} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

Herança e serialização

Quando um objeto é construído com new acontece o seguinte:

- 1) Todas as variáveis de instância recebem os valores-padrão;
- 2) O construtor é chamado, o que imediatamente chama o construtor da superclasse;
- 3) Todos os construtores de superclasse finalizam;
- 4) As variáveis de instância que são inicializadas;
- 5) O construtor finaliza

Mas tudo isso não acontece quando um objeto é deserializado. Quando uma instância de uma classe serializável é deserializada o construtor não roda, e as variáveis de instância não recebem seus valores iniciais.

Na verdade, todo o construtor acima do construtor da primeira classe não serializável rodará.

Outra coisa importante a saber é que as variáveis estáticas nunca são salvas como parte do estado do objeto, porque não pertencem ao estado do objeto.

Data, número e moeda

- ***java.util.Date***: A maior parte dos métodos desta classe foram depreciados, mas você pode usá-la como uma ponte entre as classes Calendar e DateFormat. Uma instância de Date representa uma data e hora mutáveis, até a resolução de milissegundos.

Exemplo:

```

package com.flexxo;

import java.util.Date;

public class ExemploDate {

    public static void main(String[] args) {
        //obtem a data hora de agora
        Date a = new Date();
        System.out.println(a);
        // soma a quantidade de milisegundos
        //passada por parametro na data 01/01/1970
        Date b = new Date(2000000000000L);
        System.out.println(b);
    }
}

```

A saída será (no dia/hora/minuto/segundo em que estou fazendo esta apostila):

```

Sun Mar 27 19:35:17 BRT 2011
Wed May 18 00:33:20 BRT 2033

```

- **java.util.Calendar:** Esta classe fornece uma grande variedade de métodos que lhe ajudam a converter e manipular datas e horas.

Exemplo:

```

package com.flexxo;

import java.util.Calendar;
import java.util.Date;

public class ExemploCalendar {

    public static void main(String[] args) {
        //obtem a data hora de agora
        Date a = new Date();
        // obtem uma instancia de calendar
        Calendar c = Calendar.getInstance();
        // seta a data da primeira variável
        // em milisegundos na variável Calendar
        c.setTime(a);
        if(c.MONDAY == c.getFirstDayOfWeek()){
            System.out.println("Segunda é o primeiro dia
da semana");
        }
        System.out.println("Este dia é o " +
c.get(Calendar.DAY_OF_WEEK)+ " da semana.");
    }
}

```

- **java.text.DateFormat:** Esta classe é utilizada para formatar datas em vários estilos e para vários locais do mundo.

```

package com.flexxo;

import java.text.DateFormat;

```

```
import java.util.Date;

public class ExemploDateFormat {

    public static void main(String[] args) {
        //obtem a data hora de agora
        Date a = new Date();
        DateFormat df [] = new DateFormat[6];
        df[0] = DateFormat.getInstance();
        df[1] = DateFormat.getDateInstance();
        df[2] = DateFormat.getDateInstance(DateFormat.SHORT);
        df[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
        df[4] = DateFormat.getDateInstance(DateFormat.LONG);
        df[5] = DateFormat.getDateInstance(DateFormat.FULL);

        for (DateFormat dateFormat : df) {
            System.out.println(dateFormat.format(a));
        }
    }
}
```

A saída será:

```
27/03/11 19:48
27/03/2011
27/03/11
27/03/2011
27 de Março de 2011
Domingo, 27 de Março de 2011
```

- **java.text.NumberFormat:** Esta classe é usada para formatar números e moedas para diversos locais em todo o mundo.

Exemplo:

```
package com.flexxo;

import java.text.NumberFormat;

public class ExemploNumberFormat {

    public static void main(String[] args) {
        //obtem a data hora de agora
        float f1 = 12345.6789f;
        NumberFormat df [] = new NumberFormat[3];
        df[0] = NumberFormat.getInstance();
        df[1] = NumberFormat.getCurrencyInstance();
        df[2] = NumberFormat.getNumberInstance();

        for (NumberFormat numberFormat : df) {
            System.out.println(numberFormat.format(f1));
        }
    }
}
```

A saída será:

```
12.345,679
```


R\$ 12.345,68
12.345,679

- **java.util.Locale:** Esta é usada em conjunto com DateFormat e NumberFormat para formatar datas, números e moedas para locais específicos.

Exemplo:

```
package com.flexxo;

import java.text.NumberFormat;
import java.util.Locale;

public class ExemploNumberFormat {

    public static void main(String[] args) {
        //obtem a data hora de agora
        float f1 = 12345.6789f;
        Locale lIt = new Locale("it");
        Locale lSu = new Locale("it", "CH");

        NumberFormat df [] = new NumberFormat[6];
        df[0] = NumberFormat.getInstance(lIt);
        df[1] = NumberFormat.getInstance(lSu);
        df[2] = NumberFormat.getCurrencyInstance(lIt);
        df[3] = NumberFormat.getCurrencyInstance(lSu);
        df[4] = NumberFormat.getNumberInstance(lIt);
        df[5] = NumberFormat.getNumberInstance(lSu);

        for (NumberFormat numberFormat : df) {
            System.out.println(numberFormat.format(f1));
        }
    }
}
```

A saída será:

```
12.345,679
12'345.679
¤ 12.345,68
SFr. 12'345.68
12.345,679
12'345.679
```

Parsing, Tokenização e formatação

Imagine que você possui uma grande quantidade de texto onde precisa procurar algo. Para isto é necessário maneiras fáceis de para pesquisar. Utilizaremos as classes `java.regex.Pattern`, `java.regex.Matcher` e `java.util.Scanner` para nos auxiliar e os comandos `printf` e `format` para formatar.

Introdução a buscas

Independente da linguagem que esteja usando, mais cedo ou mais tarde você terá a necessidade de realizar buscas em grandes quantidades de dados textuais, procurando por coisas específicas. As expressões regulares são um tipo de linguagem dentro da linguagem, elaboradas para ajudar os programadores em tarefas de busca. Toda a linguagem que fornece funções regex usa um ou mais motores regex. Os motores regex fazem buscas através de dados textuais usando instruções que são programadas em expressões. Uma expressão regex funciona como um pequeno programa ou script, quando você chama um motor, você passa para ele os dados textuais, e passa também a expressão que deseja usar para procurar os dados.

Buscas simples

Para o nosso primeiro exemplo, gostaríamos de buscar a String fonte:

abaaaba

Por todas as ocorrências da expressão:

ab

Código:

```
package com.flexxo;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ExemploRegexSimples {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("ab");
        Matcher m = p.matcher("abaaaba");
        while(m.find()){
            System.out.print(m.start()+ " ");
        }
    }
}
```

E o resultado será:

0 4

Ou seja, o motor encontrou correspondências na posição 0 e 4 da String fornecida. Em geral uma busca regex roda da esquerda para direita e, depois que um caractere fonte tenha sido usado em uma ocorrência, ele não pode ser reutilizado.

Metacaracteres

Em regex, alguns símbolos são utilizados para significar grupos de caracteres ou caracteres sozinhos, exemplo:

- \d – Dígito

- \s – Espaço em branco
- \w – Letras, dígitos ou underscores
- . – Qualquer caractere

Exemplo:

```
Pattern p = Pattern.compile("\\d");
Matcher m = p.matcher("a 12 b 34 c");
while(m.find()){
    System.out.print(m.start()+ " ");
}
```

A saída será:

2 3 7 8

Quantificadores

Quando estamos buscando por algo cuja extensão seja variável, obter apenas uma posição inicial não é o suficiente. Precisamos identificar uma sequência de padrões dentro de determinado texto. Os quantificadores mais utilizados são:

- + : 1 ou mais ocorrências
- * : 0 ou mais ocorrências
- ? : Zero ou uma ocorrência

Utilizando o mesmo exemplo anterior acrescentando o quantificador +, podemos notar a diferença na saída.

```
//Podemos ler a expressão abaixo como:
//Encontre um ou mais dígitos seguidos
Pattern p = Pattern.compile("\\d+");
Matcher m = p.matcher("a 12 b 34 c");
while(m.find()){
    System.out.print(m.start()+ " ");
}
```

A saída será:

2 7

Tokenização

A tokenização é o processo de dividir grandes pedaços de dados em pequenos e armazenar estes pequenos em variáveis. Provavelmente a situação mais comum de tokenização é a leitura de um arquivo delimitado.

Exemplo de tokenização com String.split():

```
String s = "1,2,3,4,5,6";
String array [] = s.split(",");
for (String string : array) {
    System.out.println(string);
}
```

```
}
```

E a saída será:

```
1
2
3
4
5
6
```

Exemplo idêntico ao acima, porém utilizando Scanner (a saída será a mesma):

```
Scanner sc = new Scanner("1,2,3,4,5,6");
sc.useDelimiter(",");
while (sc.hasNext()) {
    if (sc.hasNextInt()) {
        int i = sc.nextInt();
        System.out.println(i);
    }
}
```

Formatando com printf e format

Os métodos printf e format são idênticos, desta forma tudo o que falarmos de um é verdade para o outro. Inicialmente precisamos saber a sintaxe dos comandos :
printf("string de formatação", argumento (s));

A string de formatação pode conter tanto informações de string literais quanto dados de formatação específicos. A dica para determinar se você está olhando para dados de formatação é que estes sempre começam com %.

Exemplo:

```
System.out.printf("%2$d + %1$d", 123, 456);
```

E a saída será:

```
456 + 123
```

Dentro das aspas duplas há uma string de formatação, depois um + e depois uma segunda string de formatação. A sintaxe das strings de formatação é:

% [arg_index\$] [flags] [width] [.precision] conversion char

- ***arg_index***: Um número inteiro seguido diretamente por um cifrão, isso indica qual argumento deverá ser exibido nesta posição.
- ***flags***: São símbolos que significam algo. Abaixo os mais usados:
 - ***"-"***: Justifica o argumento a esquerda
 - ***"+"***: Inclui um sinal (+ ou -) com este argumento
 - ***"0"***: Preenche os vazios do argumento com zeros.
 - ***","***: Usa separadores de agrupamentos específicos do local.

- “(“: Coloca números negativos entre parênteses.
- **width:** Este valor indica o número mínimo de caracteres a serem exibidos.
- **precision:** Indica o número de dígitos a serem exibidos na parte fracionária.
- **conversion:** O tipo do argumento que está sendo formatado:
 - b – boolean
 - c – char
 - d – integer
 - f – float ou double
 - s – String



Genéricos e conjuntos



- 1) hashCode() e equals()
- 2) Conjuntos
- 3) Tipos genéricos

hashCode() e equals()

Sobrescrevendo equals()

O objetivo do método equals é definir se dois objetos são significativamente iguais, ou seja, se o conteúdo de seus atributos de instância, são iguais. Para determinarmos isto basta sobrescrevermos o método abaixo, que é herdado da classe Object, na classe que tivermos a necessidade de efetuar a comparação entre objetos.

```
public boolean equals(Object obj)
```

A não implementação deste método poderá implicar em:

- Não poder comparar objetos da classe deste tipo;
- Não poder usar o objeto como a chave em uma tabela hashing;
- Não obter Sets precisos, ou seja, sets sem objetos duplicados;

Exemplo:

```
package com.flexxo;

class Dinheiro {
    private String desDinheiro;
    private String desSimbolo;
    private String desPais;

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

    public String getDesDinheiro() {
        return desDinheiro;
    }
    public void setDesDinheiro(String desDinheiro) {
        this.desDinheiro = desDinheiro;
    }
    public String getDesSimbolo() {
        return desSimbolo;
    }
    public void setDesSimbolo(String desSimbolo) {
        this.desSimbolo = desSimbolo;
    }
    public String getDesPais() {
        return desPais;
    }
    public void setDesPais(String desPais) {
        this.desPais = desPais;
    }
}

public class ExemploEquals {
    public static void main(String[] args) {
        Dinheiro d1 = new Dinheiro();
        d1.setDesDinheiro("Reais");
        Dinheiro d2 = new Dinheiro();
        d2.setDesDinheiro("Dolar");
        if(d1.equals(d2)){
            System.out.println("iguais");
        }else{
            d2.setDesDinheiro("Reais");
            if(d1.equals(d2)){
                System.out.println("Dinheiros iguais");
            }
        }
    }
}
```

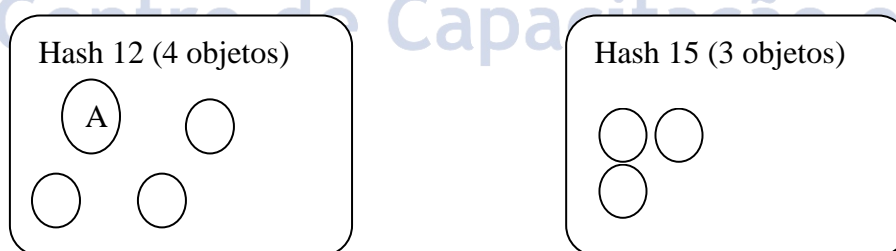
O contrato equals()

Para sobrescrevermos o método equals de forma adequada, devemos considerar as seguintes regras na sua implementação:

- *É reflexivo*: Para qualquer valor de referência `x.equals(x)` o resultado deve ser true;
- *É simétrico*: Para qualquer valor de referência `x` e `y`, `x.equals(y)` deve retornar true, se e somente se, `y.equals(x)`;
- *É transitivo*: Para qualquer valor de referência `x`, `y` e `z`, se `x.equals(y)` retornar true e `y.equals(z)` também retornar true, então `x.equals(z)` deve retornar true;
- *É consistente*: Para qualquer valor de referência `x` e `y`, múltiplas chamadas de `x.equals(y)` retornarão consistentemente true ou consistentemente false, contanto que nenhuma informação usada nas comparações do objeto de equals tenha sido alterada;
- Para qualquer valor de referência `x` que não seja null, `x.equals(null)` deve retornar false.

Sobrescrevendo hashCode()

O código hashing normalmente é usado para melhorar o desempenho de grandes conjuntos de dados. O valor de hashing de um objeto é usado por algumas classes de conjuntos (HashMap e HashSet). O código hashing é como se fosse um código agrupador o qual determina em que conjunto de objetos o objeto está. No exemplo abaixo temos 4 objetos com o valor do hash 12 e 3 objetos de valor 15. No momento que um conjunto HashMap ou HashSet procurar por estes objetos, primeiro ele irá buscar em qual conjunto deve procurar, para depois procurar o objeto que precisa, ou seja, neste caso, se estivesse procurando pelo objeto A, 3 objetos não entrariam na busca, o que aumentaria a performance.



Assinatura:

```
public int hashCode()
```

Exemplo:

```
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result  
        + ((desDinheiro == null) ? 0 :  
desDinheiro.hashCode());  
    return result;  
}
```

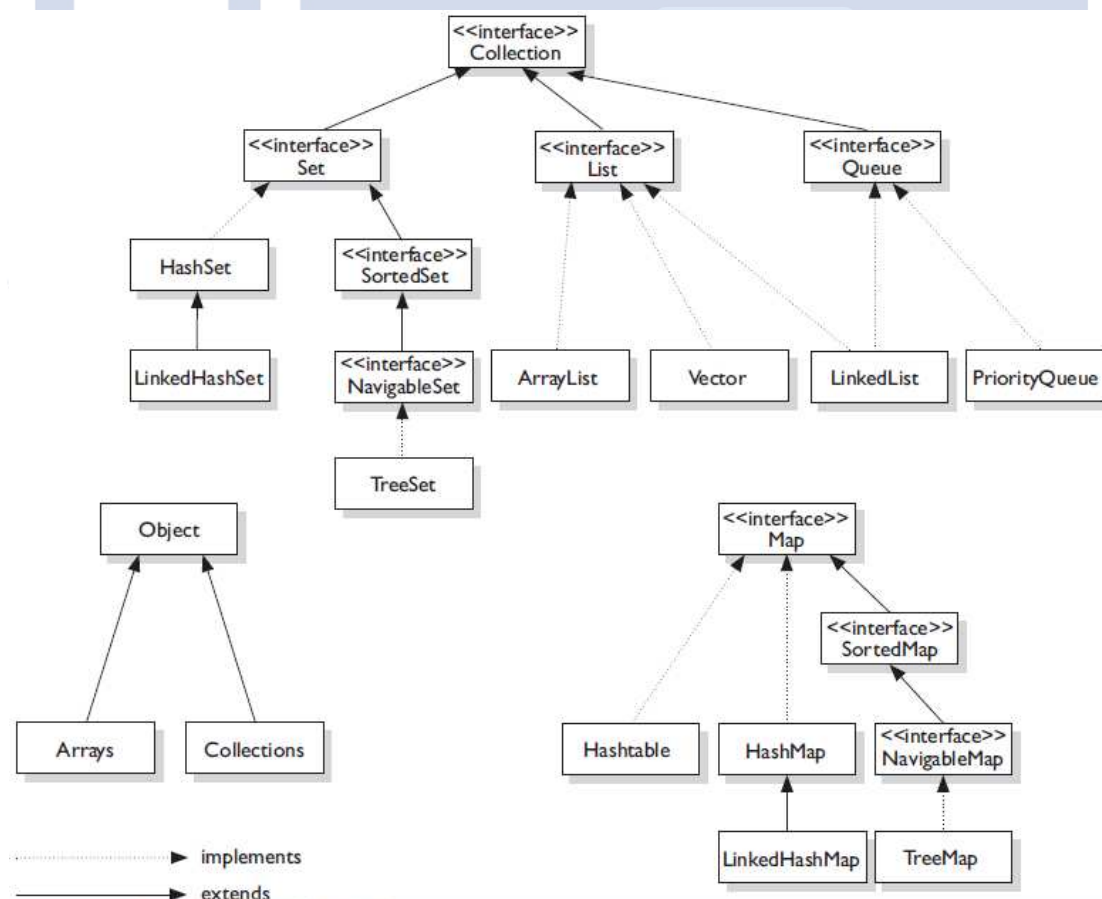

O contrato hashCode()

Assim como o método equals(), o método hashCode() possui algumas regras para sua implementação correta, são elas:

- *Consistente*: Mesmo que seja chamado n vezes, este objeto terá que retornar o mesmo valor, desde que nenhuma informação tenha sido alterada entre as chamadas.
- Se dois objetos forem iguais com o método equals(), então o retorno do método hashCode() nos dois objetos deve ser o mesmo.
- Não é obrigatório que quando dois objetos forem diferentes de acordo com o método equals, a chamada do método hashCode produza valores distintos.

Conjuntos

A estrutura de conjuntos da linguagem fornece listas, conjuntos, mapas e filas que satisfarão a maioria das necessidades na codificação. Abaixo podemos verificar um diagrama de classes dos conjuntos disponíveis na versão Java 6.



Há algumas operações básicas que normalmente são usadas em conjuntos:

- Adicionar objetos ao conjunto;
- Remover objetos do conjunto;

- Saber se um objeto está no conjunto;
- Recuperar um objeto (sem removê-lo);
- Iterar pelo conjunto;

Conjuntos classificados e ordenados

Quando um conjunto é do tipo ordenado, significa que você pode iterar em seus elementos em uma ordem específica. Já para os tipos classificados significa que a ordem do conjunto já está determinada por alguma regra, esta conhecida como ordem de classificação.

A interface List

Em conjuntos do tipo List, o índice é o destaque deste tipo. Por este motivo, existem vários métodos relacionados ao índice podem ser verificados: `get(int índice)`, `indexOf()`, `add(int índice, Object obj)`. As 3 implementações desta interface são ordenadas através do índice.

- **ArrayList**: Considere esse conjunto como um array que pode crescer. Ele proporciona iteração e acesso aleatório com rapidez. Este é um conjunto não-classificado. Este conjunto é ideal quando for necessário iteração rápida e não necessitar de muitas operações de inserção e exclusão.
- **Vector**: Este conjunto é idêntico ao ArrayList, porém seus métodos são sincronizados.
- **LinkedList**: Este conjunto também é ordenado pelo índice, assim como o ArrayList, exceto pelos seus elementos serem duplamente encadeados. Este encadeamento fornece novos métodos, além dos métodos da interface, para inserção ou remoção do início ou final, o que o torna uma opção adequada à implementação de uma pilha ou fila. Apesar de ser mais lento que ArrayList, este conjunto é indicado quando houver a necessidade de inclusões ou remoções rápidas.

A interface Set

O conjunto Set dá importância a exclusividade, ou seja, não é possível ter objetos duplicados. O método `equals()` determinará se dois objetos são idênticos.

- **HashSet**: Este é um conjunto não-classificado e não-ordenado. Ele usa o código hashing do objeto que está sendo inserido, desta forma quanto mais for eficiente a implementação do método `hashCode()` melhor será o desempenho no acesso. Quando o objetivo for um conjunto sem duplicatas e sem necessidade de ordem de iteração, utilize este conjunto.
- **LinkedHashSet**: Este é uma versão ordenada de HashSet que mantém uma lista duplamente encadeada para todos os elementos. Este conjunto mantém a ordenação pela ordem em que forem sendo inseridos os objetos. Use esta classe ao invés de HashSet, quando a ordem de iteração for importante.
- **TreeSet**: Este é um conjunto classificado, garantido que seus elementos fiquem em sequência ascendente, de acordo com a sua ordem natural.

A interface Map

Para Maps os identificadores exclusivos é que são relevantes. Uma chave exclusiva é mapeada para um valor específico, sendo que, tanto a chave quanto o valor são objetos.

- *HashMap*: Este é um conjunto não-classificado e não-ordenado. Quando houver a necessidade de um conjunto Map sem necessitar ordem de iteração esta é a melhor opção. Assim como HashSet, este conjunto usa o código hashing do objeto que está sendo inserido. Permite uma chave e diversos valores null.
- *Hashtable*: É a versão sincronizada do HashMap, porém não permite valores ou chaves nulas.
- *LinkedHashMap*: Assim como o LinkedHashSet, este conjunto mantém a ordem de inserção. Embora seja um pouco mais lento que HashMap para inserir ou remover elementos, sua iteração é mais rápida.
- *TreeMap*: Esta é a versão Map do TreeSet.

A interface Queue

A função da Queue é armazenar coisas a fazer, ou itens a serem processados. Embora outras ordens sejam possíveis, as filas normalmente são consideradas do tipo FIFO (em que o primeiro item a entrar é o primeiro a sair).

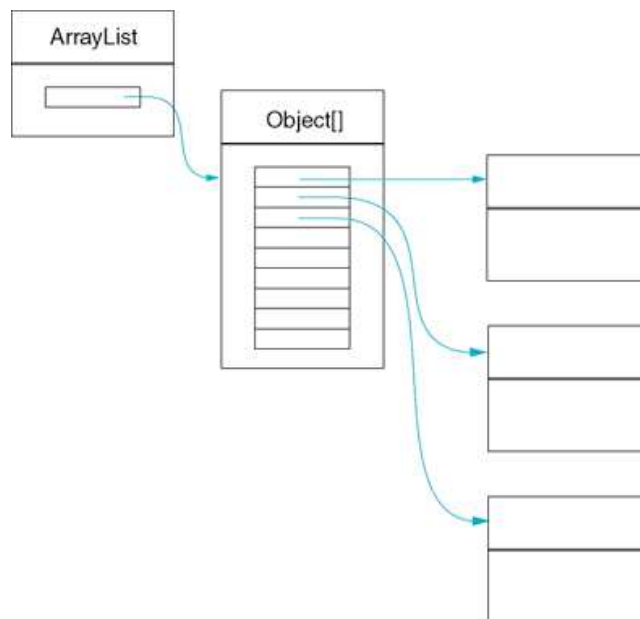
- *PriorityQueue*: O propósito de uma PriorityQueue é criar uma fila com ordenação “prioridade de entrada – prioridade de saída”, ao invés do sistema FIFO. Os elementos de PriorityQueue são ordenados ou naturalmente.

ArrayList

Um dos conjuntos mais utilizados de todos, possui algumas características importantes:

- É capaz de crescer dinamicamente;
- Fornece mecanismos de busca e inserção mais poderosos que os arrays.





Visão concreta

Exemplo:

```
ArrayList<String> listaStrings = new ArrayList<String>(0);
//Inclui no array na primeira posição (0)
listaStrings.add("posição 0");
//Inclui no array na primeira posição (1)
listaStrings.add("posição 1");
//Inclui no array na primeira posição (2)
listaStrings.add("posição 2");
// busca a posição 1
System.out.println(listaStrings.get(1));
// remove a posição 1
listaStrings.remove(1);
```

```
for (String string : listaStrings) {
    System.out.println(string);
}
```

Comparable e Comparator

A interface Comparable é utilizada quando necessitamos ordenar nossa lista de objetos ou array. O objeto que estiver dentro da lista, deverá implementar a interface Comparable e sobrescrever o método compareTo, que determinará a ordenação da collection ou array, quando chamados os métodos Collections.sort ou Arrays.sort.

O método compareTo, retorna um int com as seguintes características:

- *Negativo*: Se this.object < otherObject;
- *Zero*: Se this.object == otherObject;
- *Positivo*: Se this.object > otherObject;

Exemplo:

```
package com.flexxo;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Animal implements Comparable{
    private String desAnimal;

    public String getDesAnimal() {
        return desAnimal;
    }

    public void setDesAnimal(String desAnimal) {
        this.desAnimal = desAnimal;
    }

    @Override
    public int compareTo(Object arg0) {
        Animal a = (Animal)arg0;
        return this.getDesAnimal().compareTo(a.getDesAnimal());
    }
}

public class ExemploComparable {
    public static void main(String[] args) {
        List<Animal> lista = new ArrayList<Animal>(0);
        Animal a = new Animal();
        a.setDesAnimal("Cachorro");
        lista.add(a);
        a = new Animal();
        a.setDesAnimal("Abelha");
        lista.add(a);
        a = new Animal();
        a.setDesAnimal("Gato");
        lista.add(a);
        //Antes da ordenação
        for (Animal animal : lista) {
            System.out.println(animal.getDesAnimal());
        }
        System.out.println("-----");
        Collections.sort(lista);
        //Após a ordenação
        for (Animal animal : lista) {
            System.out.println(animal.getDesAnimal());
        }
    }
}
```

A interface `Comparator` fornece a capacidade de classificar a uma dada coleção de diversas formas. Além disso é possível utilizá-la para classificar instâncias de qualquer classe, ao contrário da `Comparable`, que lhe força a modificar a classe cujas instâncias você deseja classificar. A interface `Comparator` também é fácil de implementar, precisamos implementar apenas o método `compare`.

Exemplo:

```
package com.flexxo;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class OrdenacaoInversaAnimal implements Comparator<Animal> {
    @Override
    public int compare(Animal arg0, Animal arg1) {
        return
arg1.getDesAnimal().compareTo(arg0.getDesAnimal());
    }
}

class Animal{
    private String desAnimal;
    public String getDesAnimal() {
        return desAnimal;
    }
    public void setDesAnimal(String desAnimal) {
        this.desAnimal = desAnimal;
    }
}

public class ExemploComparator {
    public static void main(String[] args) {
        List<Animal> lista = new ArrayList<Animal>(0);
        Animal a = new Animal();
        a.setDesAnimal("Cachorro");
        lista.add(a);
        a = new Animal();
        a.setDesAnimal("Abelha");
        lista.add(a);
        a = new Animal();
        a.setDesAnimal("Gato");
        lista.add(a);
        // Antes da ordenação
        for (Animal animal : lista) {
            System.out.println(animal.getDesAnimal());
        }
        System.out.println("-----");
        Collections.sort(lista, new OrdenacaoInversaAnimal());
        // Após a ordenação
        for (Animal animal : lista) {
            System.out.println(animal.getDesAnimal());
        }
    }
}
```

Busca em Arrays e Collections

As classes Collections e Arrays fornecem métodos que lhe permitem procurar por um elemento específico. Ao fazer buscas em conjuntos ou arrays, aplicam-se as seguintes regras:

- As buscas são realizadas usando-se o método `binarySearch()`.
- As buscas bem sucedidas retornam o índice (int) do elemento sendo procurado.

- As buscas mal sucedidas retornam um índice negativo que representa o ponto de inserção. Este é o lugar onde o elemento seria inserido para manter o conjunto/array ordenado. Se o ponto de inserção for na posição 2, o método retornará -3, ou seja, posição = (- (índice retornado) - 1).
- O conjunto/array deve estar ordenado antes de efetuar a busca.
- Caso não for ordenado antes de efetuar a busca, o retorno será imprevisível.
- Se o conjunto/array foi ordenado usando-se um comparator, este também deverá ser informado no método `binarySearch`.

Exemplo:

```
String array[] = { "one", "two", "three", "four" };
Arrays.sort(array);
for (String string : array) {
    System.out.print(string + " ");
}
System.out.println();
System.out.println(Arrays.binarySearch(array, "one"));
```

A saída será:

```
four one three two
1
```

Conversão de Arrays e Lists

Existem alguns métodos que lhe permitem converter arrays em lists, e lists em arrays. As classes `List` e `Set` possuem `toArray()`, e a classe `Arrays` possui um método chamado `asList()`.

O método `Arrays.asList()` copia um array para dentro de um `List`. A API diz, “retorna uma lista de tamanho fixo baseada no array especificado (modificações feitas na lista retornada são escritas diretamente no array)”. Quando utiliza-se o método `asList()`, o array e o list ficam unidos como se fossem um só. Quando altera-se um deles, o outro é atualizado automaticamente.

Exemplo:

```
String array [] = { "a", "b", "c", "d" };
List<String> lista = Arrays.asList(array);
for (String s : lista) {
    System.out.println(s);
}
array[0] = "abcde";
for (String s : lista) {
    System.out.println(s);
}
```

Usando Sets

Abaixo alguns exemplos do uso de Sets:

- *Set:*

```
Set<String> a = new HashSet<String>();
```

```
//insere a no set retorna true
boolean inseriu = a.add("a");
System.out.println(inseriu);
//tenta inserir a denovo retorna false
inseriu = a.add("a");
System.out.println(inseriu);
//tenta remover um registro não existente retorna false
boolean removeu = a.remove("b");
System.out.println(removeu);
//remove a retorna true
removeu = a.remove("a");
System.out.println(removeu);
```

- *TreeSet:*

```
Set<String> a = new TreeSet<String>();
a.add("d");
a.add("a");
a.add("e");
a.add("f");
//TreeSet mantém a ordem natural
for (String string : a) {
    System.out.println(string);
}
```

Usando Maps

Abaixo alguns exemplos do uso de Maps:

- *HashMap:*

```
Map<String, Integer> a = new HashMap<String, Integer>();
//primeiro parametro é a chave
//segundo é o valor
a.put("b", 1);
a.put("a", 5);
a.put("z", 3);
//imprime 5
System.out.println(a.get("a"));
```

- *TreeMap:*

```
Map<String, Integer> a = new TreeMap<String, Integer>();
//primeiro parametro é a chave
//segundo é o valor
a.put("b", 1);
a.put("a", 5);
a.put("z", 3);
//imprime {a=5, b=1, z=3}
System.out.println(a);
```

Usando Queue

Abaixo um exemplo do uso de Queue:

```
PriorityQueue<Integer> a = new PriorityQueue<Integer>();
//empilha
```



```

a.offer(1);
//empilha
a.offer(2);
//empilha
a.offer(3);

//desempilha
System.out.println(a.poll());
//apenas retorna retorna
//o elemento sem desempilhar
System.out.println(a.peek());
//desempilha
System.out.println(a.poll());

```

Tipos genéricos

Antes de java 5 não havia nenhuma sintaxe para declarar um conjunto com esse mesmo comportamento. Para criar um ArrayList de Strings, utilizava-se:

```
ArrayList a = new ArrayList();
```

Ou equivalente polimórfico:

```
List a = new ArrayList();
```

Não havia nenhuma sintaxe que lhe permitisse especificar que uma lista utilizaria Strings e somente Strings. Desta forma, o compilador não podia forçá-lo a colocar apenas coisas do tipo especificado na lista. A partir do java 5 podemos usar genéricos, que nada mais é que especificar um tipo para um determinado conjunto.

Exemplo:

```
List<String> listaTipada = new ArrayList<String>();
listaTipada.add("ABC");
```

Coringa

Existe um mecanismo que diz ao compilador que se pode utilizar qualquer subtipo genérico do tipo do argumento declarado, este mecanismo chama-se coringa <?>, geralmente utilizado em parâmetros de métodos.

Exemplo (com extends):

```

public static void getParametro(List<? extends Number> lista){
    //Quando utiliza-se o extends, não é possível
    //adicionar elementos a lista
    System.out.println(lista);
}

public static void main(String[] args) {
    List<Integer> lista = new ArrayList<Integer>();
    getParametro(lista);
}

```

Exemplo (com super):

```
public static void getParametro(List<? super Integer> lista){
    //Já se utilizarmos o super é possível
    //adicionar elementos na lista
    lista.add(1);
}

public static void main(String[] args) {
    List<Integer> lista = new ArrayList<Integer>();
    getParametro(lista);
}
```

Declarações genéricas

É possível declararmos as nossas próprias classes como tipos genéricos. Podemos criar uma classe que requeira que seja passado um tipo ao declará-la ou instanciá-la.

Exemplo:

```
public interface List<E>
```

O <E> é um marcador para o tipo que você passará. A interface List é um modelo, ao escrever o seu código, você modifica o List genérico para List<Number> ou List<Integer>. O E é apenas uma convenção, qualquer identificador válido funcionaria, porém E (“*elemento*”) geralmente é utilizado quando o modelo é um conjunto. A outra convenção é T (“*tipo*”), quando o modelo for qualquer coisa diferente de conjunto.

Exemplo do método:

```
public boolean add(E o)
```

Métodos genéricos

Além da possibilidade de usar o tipo declarado com o nome da classe, podemos definir um tipo parametrizado em nível de método. Desta forma, a classe em si não precisará ser genérica.

Exemplo:

```
public <T> boolean add(T o){
    List<T> l = new ArrayList<T>(0);
    l.add(o);
    return true;
}
```

Classes internas



- 1) Classe interna comum
- 2) Locais de método
- 3) Anônimas
- 4) Estáticas aninhadas

Às vezes nos vemos projetando uma classe na qual descobrimos ser preciso um comportamento que pertence a uma classe separada especializada, mas que também terá de estar associada a classe que estivermos projetando.

Métodos responsáveis pela manipulação de eventos talvez sejam melhor exemplo disto.

Um dos benefícios das classes internas é o relacionamento especial que a instância de uma classe interna compartilha com uma instância da classe externa. Esse relacionamento especial, concede ao código da classe interna acesso aos membros da classe que a estiver encapsulando, como se a classe interna fizesse parte da externa, ou seja, a classe interna é um membro da externa. A instância da interna terá acesso a todos os membros da classe externa, inclusive os privados.

Comum

Usamos o termo comum para representar classes internas que não sejam: estáticas, locais de método e anônimas.

Exemplo:

```
package com.flexxo;

public class ExemploClasseInternaComum {

    class Interna{

    }

}
```

A classe interna continua sendo uma classe separada, pois ao compilarmos a classe acima teremos 2 arquivos .class: *ExemploClasseInternaComum.class* e *ExemploClasseInternaComum\$Interna.class*. Entretanto não é possível acessá-la diretamente executando `java ExemploClasseInternaComum$Interna`, esperando que o método main da classe *Interna* seja executado. A única forma de acessar uma classe interna é através da instância da classe externa.

Exemplo:

```
package com.flexxo;

public class ExemploClasseInternaComum {
    private int x = 10;
    class Interna{
        Interna(){
            System.out.println(x);
        }
    }
    public static void main(String[] args) {
        ExemploClasseInternaComum a = new
        ExemploClasseInternaComum();
        a.new Interna();
    }
}
```

Locais de método

Além das classes internas comuns, também é possível declará-las dentro dos métodos. Vejamos um exemplo a seguir:

Exemplo:

```
package com.flexxo;

public class ExemploClasseLocalMetodo {
    private int x = 10;

    private void doSomething(){
        class InternaLocalMetodo{
            InternaLocalMetodo(){
                System.out.println(x);
            }
        }
        InternaLocalMetodo a = new InternaLocalMetodo();
    }

    public static void main(String[] args) {
        ExemploClasseLocalMetodo a = new
ExemploClasseLocalMetodo();
        a.doSomething();
    }
}
```

Anônimas

Até agora examinamos a definição de uma classe dentro de outra classe encapsuladora e dentro de um método. Outra forma de implementarmos uma classe interna são as classes anônimas. Vejamos um exemplo:

```
package com.flexxo;

class Cachorro {
    public void latir(){
        System.out.println("au au au");
    }
}

public class ExemploClasseAnonima {

    public static void main(String[] args) {
        //a linha abaixo apos a chave
        //refere-se a implementacao de uma classe
        //anonima
        Cachorro chowchow = new Cachorro(){
            //aqui redefinimos o metodo latir
            public void latir(){
                System.out.println(" row row row ");
            }
        };
    }
}
```

```
        chowchow.latir();  
    }  
}
```

Estáticas aninhadas

Esta é um tipo de classe não-interna ou de “nível superior”, cujo escopo se encontra dentro de outra classe. Esta nada mais é que um membro estático da classe encapsuladora.

Exemplo:

```
package com.flexxo;  
  
public class ExemploClasseEstatica {  
    static class ClasseInterna{  
        public void doSomething(){  
            System.out.println("Faz algo");  
        }  
    }  
    public static void main(String[] args) {  
        ExemploClasseEstatica.ClasseInterna a = new  
        ExemploClasseEstatica.ClasseInterna();  
        a.doSomething();  
    }  
}
```

Flexxo

Centro de Capacitação em TI

Threads



- 1) Iniciando Threads
- 2) Estados e transições
- 3) Sincronização
- 4) Interação entre Threads



Imagine um aplicativo para corretores do mercado financeiro com vários comportamentos complexos iniciados pelo usuário. Uma das aplicações seria “fazer o download das últimas opções de preços das ações”, uma segunda seria “verificar avisos de preços” e uma terceira operação mais demorada, seria “analisar dados históricos da empresa XYZ”.

Em um ambiente de execução de segmentação única, essas ações serão executadas uma após a outra, ou seja, a próxima ação só pode executar quando a anterior tiver sido concluída. Se a análise de um histórico levar meia hora e o usuário selecionar a execução do download seguido da pesquisa, o aviso de compra ou venda pode vir tarde demais.

Neste caso o ideal seria que o download ocorresse em segundo plano (outro thread). Desta forma, outros processos poderiam ocorrer ao mesmo tempo para que um aviso pudesse ser comunicado instantaneamente. Assim o usuário ficaria livre para interagir com outras partes do aplicativo. A análise também poderia ocorrer em um thread separado, para que o usuário pudesse trabalhar em outros locais do aplicativo enquanto que os resultados estivessem sendo calculados.

Uma instância de Thread é apenas um objeto. Quando está em execução é um processo individual que possui suas próprias pilhas de chamada. O método main, por exemplo, que dá início a todo o processamento, é um thread e é chamado de thread principal.

Iniciando threads

É possível iniciar e instanciar um Thread de duas maneiras:

- Estendendo a classe `java.lang.Thread` (sobrescrevendo o método `run()`)
- Implementando a interface `Runnable` (implementando o método `run()`)

Toda a ação começa no método `run()`. Em uma frase, no método `run` estará “todo o trabalho a ser feito”.

Exemplo estendendo a classe Thread:

```
package com.flexxo;

public class ExemploThread extends Thread {

    public void run() {
        System.out.println("Thread independente");
    }

    public static void main(String[] args) {
        ExemploThread a = new ExemploThread();
        a.start();
    }
}
```

Exemplo implementando a interface Runnable:

```
package com.flexxo;

public class ExemploThread implements Runnable {

    public void run() {
        System.out.println("Thread independente");
    }

    public static void main(String[] args) {
        ExemploThread t = new ExemploThread();
        Thread a = new Thread(t);
        a.start();
    }
}
```

Estados e transições

O Thread só pode ficar em um dos cinco estados abaixo:

- *Novo*: Este é o estado em que o Thread se encontra depois que a instância de Thread foi criada, mas o método `start()` não foi chamado. Ele se torna um objeto Thread ativo, mas ainda não é um Thread em execução. Nesse momento é considerado inativo.

- **Executável:** Esse é o estado em que um Thread se encontra quando está qualificado para ser executado, mas o agendador não o selecionou como Thread a ser processado. O Thread entra pela primeira vez no estado executável quando o método `start()` é chamado, mas também pode retornar ao estado executável depois de ser processado ou ao retornar de um estado bloqueado, de espera ou suspensão. Quando o thread está no estado executável é considerado ativo.
- **Execução:** Esse é o estado em que o thread se encontra quando o agendador o seleciona. Um thread pode sair de um estado de execução por vários motivos.
- **Espera/bloqueio/suspensão:** Este é o estado de um thread quando está qualificado para executar. Na verdade são 3 três estados, porém possuem um ponto em comum: é o estado em que o thread ainda está ativo, porém não está qualificado para execução.
- **Inativo:** Um thread é considerado inativo quando seu método `run()` é concluído. Um thread nunca poderá ser ativado novamente. Se você chamar `start()` em uma instância inativa de Thread, receberá uma exceção.

Método `sleep()`

O método `sleep()` é um método static da classe Thread. Use-o em seu código para “desacelerar um thread”, forçando a entrar no modo de suspensão antes de retornar ao estado executável. Quando um thread é suspenso, vai para algum local e não retorna ao estado executável até que seja despertado.

Exemplo:

```
try {  
    //Suspende a execucao por 5 minutos  
    Thread.sleep(5*60*1000);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

Método `yield()`

Para entender o método `yield()`, você terá que compreender o conceito de prioridades dos threads. Os threads são executados com algum nível de prioridade, geralmente representado por um número entre 1 e 10. Para alterarmos a prioridade de uma thread basta utilizar o método `setPriority(int)`. O agendador sempre irá optar por threads com maior prioridade. O método `yield()` retorna o thread que está sendo executado para o estado executável, a fim de permitir que outros threads com a mesma prioridade tenham a oportunidade de serem processados.

Exemplo:

```
//libera a execucao  
Thread.yield();  
//a partir daqui threads de  
//mesma prioridade terao chance  
//de serem executadas
```

Método join()

O método não-static join() da classe Thread permite que um thread seja adicionado ao final de outro Thread. Ao chamar o método join, o Thread que chamou irá executar e até este não acabar, o processo não continua.

Exemplo:

```
ExemploThread t = new ExemploThread();
Thread a = new Thread(t);
a.start();
try {
    a.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Sincronização

Como a sincronização de objetos funciona ? Com bloqueios. Todo o objeto Java possui um bloqueio interno que só surge quando o objeto tem um código de método sincronizado. Quando entramos em um método sincronizado não-estático, automaticamente adquirimos o bloqueio associado com a instância atual da classe cujo código estamos executando (a instância this).

Já que só há um bloqueio por objeto, se um thread usar o bloqueio, nenhum outro poderá acessar o código sincronizado até que o bloqueio seja liberado. Isso significa que nenhum outro thread pode entrar nos métodos sincronizados deste objeto. Sincronizando um método, você garante que quaisquer threads que tentem executar esse método usando a mesma instância do objeto, sejam impedidos de ter acesso simultâneo.

Exemplo:

```
package com.flexxo;

public class ExemploSynchronized {

    public synchronized void metodoSincronizado() {
        System.out.println("abc");
    }

    public void metodoComTrechoSincronizado() {
        synchronized(this) {
            System.out.println("abc");
        }
    }
}
```

Interação entre threads

A classe Object possui três métodos – wait(), notify() e notifyAll(), que ajudam os threads a obterem informações sobre o status de um evento. Em outras palavras, os métodos wait () e notify() permitem que um thread fique em uma “sala de espera” até que algum outro thread o avise (notifique) para voltar a atividade.

Exemplo:

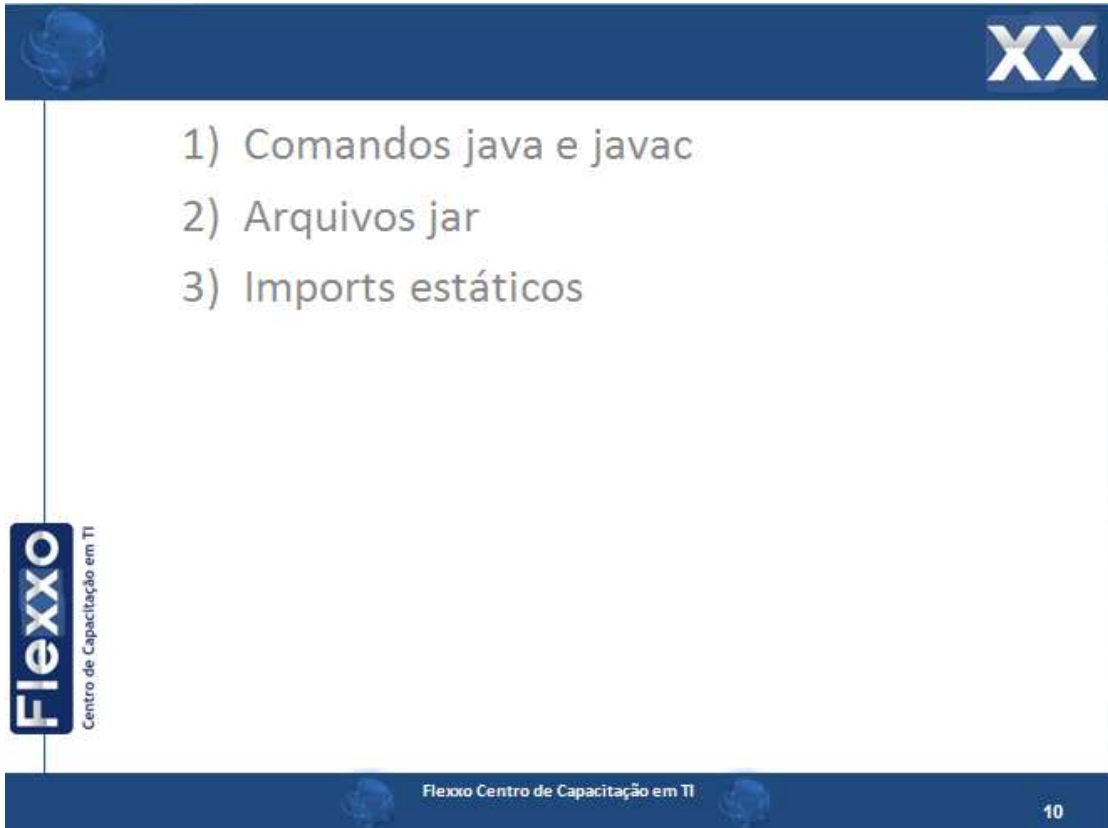
```
package com.flexxo;

class ThreadB extends Thread {
    int total = 0;

    public void run() {
        synchronized (this) {
            for (int i = 0; i < 100; i++) {
                total += i * 10;
            }
            //notifica a thread principal que
            //esta no aguardo
            notify();
        }
    }
}

public class ThreadA {
    public static void main(String[] args) {
        ThreadB b = new ThreadB();
        b.start();
        synchronized (b) {
            try {
                //ao chamar o wait, esta thread
                //principal, nao ira executar
                //ate que a chamada do notify
                //ou notifyAll seja chamado
                b.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(b.total);
        }
    }
}
```

Desenvolvimento



1) Comandos java e javac
2) Arquivos jar
3) Imports estáticos

Flexxo
Centro de Capacitação em TI

Flexxo Centro de Capacitação em TI 10

Comandos javac e java

O comando javac é usado para compilar os arquivos .java. Vejamos sua sintaxe abaixo:

```
javac [opções] [arquivos-fonte]
```

O comando java é usado para executar os arquivos compilados .class. Vejamos sua sintaxe abaixo:

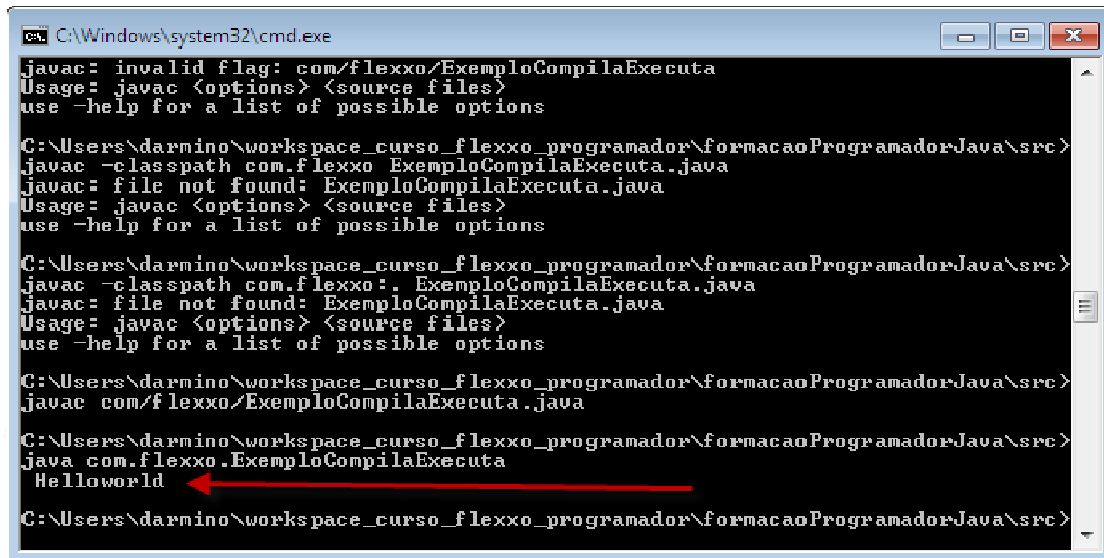
```
java [opções] class [argumentos]
```

Abaixo temos um arquivo java que escreve no console a mensagem “Helloworld”:

```
package com.flexxo;  
  
public class ExemploCompilaExecuta {  
    public static void main(String[] args) {  
        System.out.println(" Helloworld ");  
    }  
}
```

Para compilá-lo e executá-lo manualmente devemos:

- Abrir o prompt de comando;
- Se posicionar no diretório src do projeto;
- *Para compilar:* javac com/flexxo/ExemploCompilaExecuta.java
- *Para executar:* java com.flexxo.ExemploCompilaExecuta
- *E no console:* A mensagem “Helloworld” será exibida.



```
C:\Windows\system32\cmd.exe
javac: invalid flag: com/flexxo/ExemploCompilaExecuta
Usage: javac <options> <source files>
use -help for a list of possible options

C:\Users\darmino\workspace_curso_flexxo_programador\formacaoProgramadorJava\src>
javac -classpath com.flexxo: ExemploCompilaExecuta.java
javac: file not found: ExemploCompilaExecuta.java
Usage: javac <options> <source files>
use -help for a list of possible options

C:\Users\darmino\workspace_curso_flexxo_programador\formacaoProgramadorJava\src>
javac -classpath com.flexxo: ExemploCompilaExecuta.java
javac: file not found: ExemploCompilaExecuta.java
Usage: javac <options> <source files>
use -help for a list of possible options

C:\Users\darmino\workspace_curso_flexxo_programador\formacaoProgramadorJava\src>
javac com/flexxo/ExemploCompilaExecuta.java

C:\Users\darmino\workspace_curso_flexxo_programador\formacaoProgramadorJava\src>
java com.flexxo.ExemploCompilaExecuta
Helloworld
C:\Users\darmino\workspace_curso_flexxo_programador\formacaoProgramadorJava\src>
```

Compilando com -d

Ao executarmos o comando javac, por default, o compilador coloca o arquivo compilado no mesmo diretório do arquivo fonte. A opção -d (d significa destino) lhe permite dizer ao compilador em que diretório ele deverá colocar o arquivo .class gerado.

Exemplo utilizando a mesma classe de antes:

```
javac -d classes com/flexxo/ExemploCompilaExecuta.java
```

Usando propriedades de sistema

Podemos também setar variáveis de ambiente ao executar um programa Java. Estas variáveis poderão ser acessadas em qualquer lugar do programa. Para isto basta incluirmos nas opções de execução do comando java a opção -D:

Sintaxe:

-D<nome da variavel>=<valor da variavel>

Exemplo do comando:

```
java -DvarAmbiente="Variavel de ambiente" ExemploVarAmb
```

Fonte:

```
import java.util.Properties;

public class ExemploVarAmb {
    public static void main(String[] args) {
        Properties p = System.getProperties();
        System.out.println(p.getProperty("varAmbiente"));
    }
}
```

E o resultado no console:

Variavel de ambiente

Argumentos de linha de comando

Para passar um argumento via linha de comando ao programa que estamos executando, basta colocarmos eles após a classe.

Linha de comando:

```
java com.flexxo.ExemploArgsLinhaComand " Japao " " EUA "
```

Fonte:

```
public class ExemploArgsLinhaComand {
    public static void main(String[] args) {
        String parametro1 = args[0];
        System.out.println("Primeiro parametro  
="+parametro1);
        String parametro2 = args[1];
        System.out.println("Segundo parametro  
="+parametro2);
    }
}
```

Resultado:

Primeiro parametro = Japao

Segundo parametro = EUA

Usando classpaths

Classpaths, como o nome já diz “caminho das classes”, são a localização de diretórios e arquivos de um determinado programa. Em Java, determinamos o classpath a executar de uma determinada aplicação através do argumento *-classpath*.

Os caminhos são separados por : na plataforma Linux e ; na plataforma Windows.

Exemplo:

```
-classpath /com/foo/acct:/com/foo
```

Arquivos jar

Depois que você criou e testou o seu aplicativo, você poderá empacotá-lo para facilitar a distribuição e instalação do software por outras pessoas. Para isso Java fornece o arquivo JAR (Java Archive).

Para criar um arquivo JAR, basta executar o comando abaixo:

```
jar -cf <nome arquivo>.jar <diretório a ser comprimido>
```

Imports estáticos

Na versão Java 5, os imports foram melhorados possibilitando as chamadas importações estáticas. As importações estáticas podem ser usadas quando você deseja usar os membros static de uma classe.

Exemplo sem import estático:

```
package com.flexxo;

public class ExemploImportStatic {
    public static void main(String[] args) {
        System.out.println(Integer.MAX_VALUE);
    }
}
```

Exemplo com import estático:

```
package com.flexxo;
import static java.lang.System.out;
import static java.lang.Integer.*;

public class ExemploImportStatic {
    public static void main(String[] args) {
        out.println(MAX_VALUE);
    }
}
```