Guia para Consulta - Java

Resumo da API Java

Autor

Tatiana Franciely da Silva (tati.franci.silva@gmail.com): é formada como Bacharel em Análise de Sistemas pela Universidade Estadual do Centro-Oeste (Unicentro), especializada em Desenvolvimento Web pela PUC-PR e certificada em programação Java pela Sun. Trabalha com análise e programação desde 2002, utilizando a linguagem Java e Oracle (PL/SQL).

Gravata

Esse artigo tem como intuito passar o conhecimento da API Java, de um modo resumido. O alvo principal é servir como um guia de consulta para todos aqueles que utilizam o Java.

Espero que esse documento seja tão útil como foi para mim.

Introdução

Quando dei início ao estudo para minha certificação comecei a fazer um resumo com todos os capítulos necessários que eu precisava conhecer e memorizar. Como não queria ficar toda hora consultando o livro ou o site, fiz esse documento para facilitar.

Dediquei uma fase dos estudos totalmente aos simulados, onde esse resumo me ajudou a tirar dúvidas rapidamente.

Meu intuito é dividir esse documento com aqueles que queiram estudar sobre Java (seja certificação ou apenas aprendizagem).

Ressalto um detalhe a esse resumo reutilizando as palavras do Kuesley Fernandes em seu site (citado quadro Para Saber Mais):

"*Algumas referências aqui citadas, poderão também ser encontradas no livro Java 2 Certificação Sun para Programador & Desenvolvedor da Kathy Sierra e Bert Bates por isso, gostaria de deixar bem claro que esse conteúdo não foi plagiado, simplesmente quero compartilhar com outras pessoas a experiência que estou adquirindo, através do estudo dirigido pela obra de Kathy e Bert.*"

Como a citação acima, destaco que o documento não tem intuito de copiar ou plagiar nenhum autor, e sim, compartilhar e ajudar outras pessoas que gostariam de obter conhecimento sobre o Java.

1. Fundamentos da Linguagem

tipo	bits	fórmula	faixa
byte	8	-(2)^7 à 2^7 -1	-128 ~ +127
short	16	-(2)^15 à 2^15 - 1	-32768 a +32767
int	32	-(2)^31 à 2^31-1	-2147483648 a +2147483647
long	64	-(2)^63 à 2^63-1	-9223372036854775808 a +9223372036854775807
char	16	2^16-1	65535

* char - caracter Unicode de 16 bits (sem sinal).

Palavras Chaves

byte	short	static	long	char	boolean	double
float	public	private	protected	int	abstract	final
strictfp	transient	synchronized	native	void	class	interface
implements	extends	const	else	do	default	switch
case	break	continue	assert	if	goto	throws

Não esquecer: **null**, **false** e **true** são valores literais!

Valores de constantes para o **Integer**:

•**Integer.MIN_VALUE** == 0x80000000;

•**Integer.MAX_VALUE** == 0x7FFFFFFF;

2. Declaração e Controle de Acessos

A tabela abaixo mostra os modificadores e onde podem ser usados:

modificador	classe	método	var instância	var local
padrão	sim	sim	sim	não
public	sim	sim	sim	não
protected	não	sim	sim	não
private	não	sim	sim	não
static	não	sim	sim	não
final	sim	sim	sim	sim
abstract	sim	sim	não	não
strictfp	sim	sim	não	não
synchronized	não	sim	não	não
transient	não	não	sim	não
native	não	sim	não	não
volatile	não	não	sim	não

Definições de alguns modificadores:

- **volatile:** Quando uma variável de instância com esse modificador é alterada, a *thread* deverá sincronizar sua cópia com a cópia principal.
- **transient:** Esse modificador indica a JVM para não esquentar a cabeça com as variáveis *transient* quando for realizar a serialização de um objeto.
- **strictfp:** Define que os valores de ponto flutuante do método devem seguir o padrão I33754.
- **native:** Define que a implementação do método foi escrita em uma linguagem nativa com por exemplo C ou C++.

Considerações:

Modificadores de acesso

- Os modificadores de acesso: (**padrão**, **public**, **private** e **protected**) não podem NUNCA ser combinados.

Métodos

- Nunca poderá ser definido como (**transient**, **volative**)
- Um método nunca poderá ser **abstract** e **final**
- Um método nunca poderá ser **abstract** e **strictfp**
- Um método nunca poderá ser **abstract** e **native**
- Um método nunca poderá ser **abstract** e **synchronized**
- Um método nunca poderá ser **abstract** e **private**
- Um método final nunca poderá ser sobreposto
- Um método abstrato nunca poderá ser implementado
- Se um método for abstrato a classe também será

Variável de Instância

- Pode ter qualquer um dos quatro modificadores de acesso
- Podem ser (**volative**, **transient**, **final**, **static**)
- Não pode ser (**abstract**, **native**, **synchronized**, **strictfp**)

Interfaces

- Uma interface nunca herda uma classe
- Uma classe nunca herda uma interface
- Uma interface nunca implementa outra interface
- Uma classe nunca implementa outra classe
- Uma interface pode herdar várias interfaces
- Uma variável de uma interface sempre será implicitamente: (**public**, **final**, **static**)
- Um método de uma interface sempre será (**public**, **abstract**)

Lembre-se da Interface: java.lang.Runnable. Ela só tem um método: public void run();

3. Operadores e atribuições

Unary operators are: ++ -- + - ! ~ ()

Arithmetic operators are: * / % + -

Shift operators are: << >> >>>

Comparison operators are: < <= > >= instanceof == !=

Bitwise operators are: & ^ |

Short-circuit operators are: && ||

Conditional operators are: ?:

Assignment operators are: = "op="

Operador Condicional

variable = boolean expression ? value to assign if boolean expression is true : value to assign if boolean expression is false.

Deslocamento de Bits

>> deslocamento de bits à direita com sinal

Fórmula: valor desejado dividido por 2 elevados a x (onde x é a quantidade de bits a deslocar)

<< deslocamento de bits à esquerda com sinal

Fórmula: valor desejado multiplicado por 2 elevado a x (onde x é a quantidade de bits a deslocar)

>>> deslocamento de bits à direita sem sinal

sinal não é mantido, todo numero deslocado com esse sinal (mesmo que seja um número negativo) fica positivo

Exceto em um caso particular:

Deslocar uma quantidade superior à capacidade do tamanho. Por exemplo, deslocar 35 bits em uma variável do tipo int que comporta 32. O compilador fará divisão entre a 35 quantidade de bits a deslocar e 32 que é a capacidade do tipo em questão e o resto dessa divisão será a quantidade deslocada, nesse caso: 3

byte y = (byte) 8 >> 12;

12 % 8 = 4 (quatro bits a deslocar)

Se for um múltiplo de 8 (no caso do tipo byte), por exemplo, 16 ou 32 em um tipo byte, não será deslocado nenhum pois o resto da divisão será 0 Portanto nem sempre o deslocamento de bits com o operador (>>>) será positivo. Deslocar 32 bits ou 64 bits em um tipo int que armazena o valor -300 e terá o mesmo valor -300.

Operadores Bit a Bit

& e

| ou inclusivo

^ u exclusivo

Operador &

O operador & compara dois bits e será um se ambos o forem.

Exemplo: int x = 10 & 9

Resultado: 8 em decimal.

1010

1001

1000

Operador | ou inclusivo

Exemplo: int x = 10 | 9

Resultado: 11

1010

1001

1011

Operador ^ ou exclusivo

Exemplo: int x = 10 ^ 9

Resultado: 3

1010

1001

0011

4. Controle de fluxo, exceções e assertivas

Assertions

Assertivas - uso incorreto/inapropriado

1) Nunca manipula um AssertionError

Não use um catch e manipula um erro de assertiva

2) Não use assertiva para validar argumentos em métodos públicos

Você não pode garantir nada em métodos público, portanto usar assertiva nesse caso, não é uma boa prática

3) Não use assertiva para validar argumento da linha de comando

Isso é uma particularidade de um método público, mas segue a mesma regra

4) Não use assertivas que possam causar efeitos colaterais

Você não pode garantia que as assertivas sempre serão executadas, portanto o seu programa deve executar independentemente das assertivas, e nunca de forma diferente simplesmente porque as assertivas foram ativadas.

Uso Apropriado/Correto

1) Use assertiva para validar argumentos em métodos privados

Como a visibilidade é privada, você consegue detectar se alguém está fazendo algo errado.

2) Use assertiva sem condição em um bloco que presuma que nunca seja alcançado.

Se você tem um bloco que nunca seria alcançado, você pode usar uma assert false, pois assim saberia se em algum momento esse bloco está sendo alcançado.

3) Lançar um AssertionError explicitamente

4) Se um bloco switch não tiver uma instrução default, adicionar uma assertiva é considerada uma boa prática

•Habilitar assertion:

- java -ea
- java -enableassertion

•Desabilitar assertion:

- java -da
- java -disableassertion

•Habilitar assertion para um determinado pacote:

- java -ea:nome_pacote

•Habilita assertion em âmbito geral e desabilita nas classes do sistema:

- java -ea -das

•Habilitar assertion nas classes do sistema:

- java -esa
- java -enablesystemassertions

Como os *assertions* não estão sempre ligados no sistema, eles não devem ser usados para validar parâmetros de métodos públicos. Uma apropriada exceção de runtime deverá aparecer como:

- IllegalArgumentException,
- IndexOutOfBoundsException, ou
- NullPointerException.

No entanto, um *assertion* pode ser usado para validar parâmetros de métodos não públicos.

Uma expressão de *assertion* pode ser usada para checar:

- Uma constante interna: algo que o programador assume que deverá ser verdadeira em um particular ponto do programa.
- Uma constante de classe: algo que deve ser verdadeiro sobre cada instância de classe.
- Um constante fluxo de controle: verifica se um fluxo nunca irá alcançar um ponto do programa.

5. Orientação a objetos, sobreposição e substituição, construtores e tipos de retorno

Modificando o acesso de um método substituído

Você poderá alterar a visibilidade de um método substituído, mas nunca para uma visibilidade inferior à definida na superclasse, ou seja, se você tiver um método na superclasse com o modificador **protected** você não poderá substituí-lo por um método com modificador **private** (mais restritivo), mas, por exemplo, para um método público sem nenhum problema.

A ordem é:

- **private**
- **padrão**
- **protected**
- **public**

As seguintes classes *override* (sobrepõem) ambos os métodos *equals* e *hashCode*:

- java.lang.Byte
- java.lang.Integer
- java.util.Vector
- java.lang.String

A assinatura do método é o nome do método e seus tipos de parâmetros. O tipo do retorno não faz parte da assinatura.

Um método de instância declarado na subclasse sobrepõe um método acessível da superclasse com a mesma assinatura.

Se um método acessível de uma superclasse for *static*, então qualquer método com a mesma assinatura na subclasse deve também ser *static*.

Se o método de uma superclasse é publico, então o método sobreposto deve também ser público. Se for um método da superclasse é *protected*, então o método sobreposto deve ser *protected* ou público (conforme ordem citada acima).

Para os modificadores synchronized, native e strictfp implementados, o programador os pode alterar livremente nas subclasses.

Uma subclasse pode sobrepor uma implementação concreta de um método com uma declaração de método abstrato.

6. Java.Lang - a classe Math, String e Wrappers

Long

-hashcode: retorna endereço hash

-equals

Double → doubleValue, floatValue, intValue, longValue, parseDouble

String

Objetos String são inalteráveis, porém as variáveis de referências não.

Se uma nova String for criada e não for atribuída a nenhuma variável de referência, ela ficará perdida.

Se você redirecionar a referência de uma String para outra String, o objeto String anterior poderá ser perdido.

A classe String é final

Métodos da String

-concat - Adiciona uma string à outra, porém não altera a string em que o método está sendo executado.

-equalsIgnoreCase - Testa se uma string é igual a outra ignorando a diferença entre letras maiúsculas e minúsculas:

-length - Obtém o tamanho da string.

-replace - Substitui os caracteres de uma string.

-substring - Extrai uma string de outra.

-toLowerCase - Muda todas as letras que estiverem maiúsculas para letra minúscula.

-toUpperCase - Processo inverso do toLowerCase, ou seja, transforma em maiúscula todas as letras que estiverem minúsculas.

-trim - Retira espaços das extremidades de uma string.

-toString - retorna o valor da string. Como a classe String é derivada de Object esse método é substituído na classe String retornando o valor da String propriamente dita.

-equals - compara o valor de uma string com outra informada em argumento.

StringBuffer

Diferente da classe String, a classe StringBuffer pode sofrer alteração.

Métodos importantes

-append - esse método adiciona uma string ao StringBuffer

-insert - insere uma string em um StringBuffer, começando em 0 (zero)

-reverse - inverte todos os caracteres da StringBuffer.

-toString - retorna o valor do objeto StringBuffer, esse método é herdado de Object.

-equals - O método equals da classe StringBuffer não é substituído, isso significa que ele não compara valores. Compara end memória.

-hashcode – baseado no endereço interno.

Java.lang.Math

-abs: Retorna o valor absoluto de um argumento, veja suas assinaturas:

- public static int abs(int value)
- public static long abs(long value)
- public static float abs(float value)
- public static double abs(double value)

O resultado sempre será um número positivo a exceção de dois casos: Interger.MIN_VALUE ou Long.MIN_VALUE

Integer.MIN_VALUE -> -2147483648

Long.MIN_VALUE -> -9223372036854775808

Operações com tipos byte, short ou char, serão promovidos para int.

-ceil: Retorna o número "ponto flutuante inteiro" superior mais próximo.

- public static float ceil(float value)
- public static double ceil(double value)

-floor: Retorna o número ponto flutuante inteiro inferior mais próximo.

- public static float floor(float value)
- public static double floor(double value)

-max: Esse método retorna o maior número entre dois informados, sua assinatura de métodos é:

- public static int max(int a, int b)
- public static long max(long a, long b)
- public static float max(float a, float b)
- public static double max(double a, double b)

-min: Esse método retorna o menor número entre dois informados, suas assinaturas são:

- public static int min(int a, int b)
- public static long min(long a, long b)
- public static float min(float a, float b)
- public static double min(double a, double b)

-round: Arredonda um numero ponto flutuante recebido como argumento, veja suas assinaturas:

- public static int round(float a)
- public static long round(double a)

-random: O método Math.random retorna um número aleatório entre 0.0 e menor que 1.0.

- public static double random()

-sin: public static double sin(double a)

-cos : public static double cos(double a)

-tan: public static double tan(double a)

-sqrt: public static double sqrt(double a)

-toDegrees: Retorna um valor de um ângulo em graus, para isso você deve passar um ângulo em radiano.

- public static double toDegrees(double a)

-toRadians: Retorna em radiano um ângulo informado em graus.

- public static double toRadians(double a)

Algumas observações sobre a classe Math

```
double x;
float p_i = Float.POSITIVE_INFINITY;
double n_i = Double.NEGATIVE_INFINITY;
double n_a_n = Double.NaN;
if ( n_a_n != n_a_n) System.out.println("NaN é diferente de NaN");
// Será ecoada, pois NaN não é igual a nada inclusive a NaN
if (Double.isNaN(n_a_n)) System.out.println("é um NaN");
// resultado: é um NaN
x = Math.sqrt(n_i);
// Alerta geral ! será atribuído NaN para x
if (Double.isNaN(x)) System.out.println( "x é um NaN");
// Resultado: x é um NaN
```

```
System.out.println( 32 / 0 );
// Resultado: java.lang.ArithmeticException
System.out.println( 32.0 / 0.0 );
// Resultado: Infinity
System.out.println( -32.0 / 0.0 );
// Resultado: -Infinity
System.out.println( 32.0 / -0.0 );
// Resultado: -Infinity
System.out.println( -32.0 / -0 );
// Resultado: -Infinity
System.out.println( 32.0 / -0 );
// Resultado: Infinity
System.out.println( -32.0 / -0.0 );
// Resultado: Infinity
```

Quando o divisor e o dividendo forem números inteiros:

Se o divisor for 0, será lançada uma exceção.

Quando o divisor é 0.0:

SEMPRE HAVERÁ A TROCA DE SINAIS

Quando o divisor for 0:

NUNCA HAVERÁ A TROCA DE SINAIS, SEMPRE SERÁ MANTIDO O SINAL DO DIVIDENDO.

Quando o divisor for diferente de 0 e 0.0

SEMPRE HAVERÁ JOGO DE SINAL

Toda operação envolvendo números inteiro SEMPRE retornará um tipo **int**.

Duas constantes são definidas na classe java.lang.Math:

- java.lang.Math.PI - public static final double E 2.718281828459045d
- java.lang.Math.E - public static final double PI 3.141592653589793d

Wrappers

Todo tipo primitivo tem uma classe wrapper correspondente.

Métodos importantes

- valueOf(): método estático. Integer pode converter números em bases diferentes. Exceção java.lang.NumberFormatException.
- xxxValue(): não são métodos estáticos.
- parseXxx(): método estático. Exceção java.lang.NumberFormatException.
- toString().

Métodos estáticos nomeados

Classe Integer:

public static String toHexString(int i)

public static String toBinaryString(int i)

public static String toOctalString(int i)

Classe Long:

public static String toHexString(long i)

public static String toBinaryString(long i)

public static String toOctalString(long i)

Métodos que lançam a exceção NumberFormatException

- Todos os construtores das classes wrapper numéricas
- os métodos estáticos valueOf
- os métodos estáticos toString de Integer e Long
- os métodos parseXxx

Métodos estáticos: **valueOf** e **parseXXX**

- valueOf: retorna instância da classe.
- parseXxx e xxValue: retorna valor primitivo

7. Objetos e Conjuntos

Uma classe pode ser:

- **ordenada** - Uma classe é ordenada se pode ser iterada pelos seus elementos em uma ordem específica, através de um índice ou por exemplo pela ordem de inserção.
- **classificada -** Uma classe classificada, quando seus elementos estão classificados por algum critério, como por exemplo, em ordem alfabética, crescente ou cronológica etc. Toda classe classificada é ordenada, já uma classe ordenada pode não ser classificada.

List

As classes que implementam a interface List, são ordenadas por meio de um índice. Aceita elementos duplicados.

ArrayList - É uma estrutura de dados que tem com base um array. Array que pode ser alterado.

-Acesso seqüencial / aleatório

-Inserção é também extremamente rápida

Vector - Tem as mesmas características de ArrayList, porém seus métodos são sincronizados. Lembrar: o método Vector.elements retorna um Enumeration.

LinkedList - A diferença entre LinkedList e ArrayList é que os elementos de LinkedList são duplamente encadeados entre si Não há necessidade da realocação do array, visto que cada nó tem uma referência para seu sucessor e seu predecessor. É a melhor implementação de FIFO (First In First Out).

Resumindo as listas **-** O que precisamos saber é que o índice em uma lista é relevante, toda lista é ordenada, ou seja, podemos iterar em uma ordem especifica, seja ela pela ordem de inserção ou pela ordem do índice. A não se esqueça que não existe lista classificada !

Set

A unicidade é a característica fundamental dessas classes sendo necessário uma correta relação dos métodos *equals* e *hashCode* para seu funcionamento. Não aceita elementos duplicados

HashSet - é um conjunto não-ordenado e não-classificado, utiliza o código *hash* do elemento que está sendo inserido para saber em qual depósito deve armazenar. Nunca use essa classe quando precisar de uma ordem na iteração.

LinkedHashSet- conjunto ordenado e duplamente encadeado, podemos iterar pelos seus elementos em uma ordem. LinkedHashSet extends HashSet e implements Collection, Set, Cloneable e Serializable.

TreeSet - essa é uma das duas classes do framework de coleção da api Java que *é classificada e ordenada* - essa classificação se dá pela ordem natural da classe que está sendo inserida:

Classe	Ordem Natural
Byte	signed numerical
Character	unsigned numerical
Long	signed numerical
Integer	signed numerical

Short	signed numerical
Double	signed numerical
Float	signed numerical
BigInteger	signed numerical
BigDecimal	signed numerical
File	system-dependent lexicographic on pathname.
String	lexicographic
Date	chronological

Map

Faz um mapeamento de chave X valor, você tem um objeto **chave** para um objeto **valor**. O elemento de um Map é par de chave/valor.

HashMap- conjunto Map não-ordenado e não classificado. Permite a existência de chave e valores nulos.

Hashtable equivalente à HashMap com a diferença que métodos são sincronizados, e não permitem valores/chaves nulos

LinkedHashMap - é uma versão ordenada (ordem de inserção/acesso) da interface Map, registra seu sucessor e predecessor, classe duplamente encadeada. LinkedHashMap extends HashMap e implements Map, Cloneable e Serializable.

TreeMap - classe da Framework Collection Java que é classificada e conseqüentemente ordenada

Classe	**Map**	**Conjunto**	**Lista**	**Ordenada**	**Classificada**
HashMap	X			não	não
Hashtable	X			não	não
TreeMap	X			sim	sim
LinkedHashMap	X			sim	não
HashSet		X		não	não
TreeSet		X		sim	sim
LinkedHashSet		X		sim	não
ArrayList			X	sim	não
Vector			X	sim	não
LinkedList			X	sim	não

Observações:

- TreeSet implements SortedSet e TreeMap implements SortedMap.
- TreeSet e TreeMap armazenam elementos na ordem determinada pela interface *Comparator* ou *Comparable*.
- Todas as Tree implementam *Cloneable*.
- ListIterator extends Iterator.
- A interface Enumeration introduzida com o Java 1.0 prove uma maneira simples de se mover através dos elementos de um Vector ou de chaves/valores de uma Hashtable. Existem dois métodos: hasMoreElements e nextElement.
- A interface Iterator foi introduzida com o framework de coleções no Java 1.2. Existem 3 métodos: hasNext, next e remove.
- Interface ListIterator introduzida com o Java 1.2 na interface Iterator. A interface ListIterator extends a interface Iterator e declara métodos adicionais que provem capacidade de iterações para frente e para trás.
- A interface RandomAccess é uma interface de sinalização que não declara nenhum método. Vector e Array List implementam a interface RandomAccess.

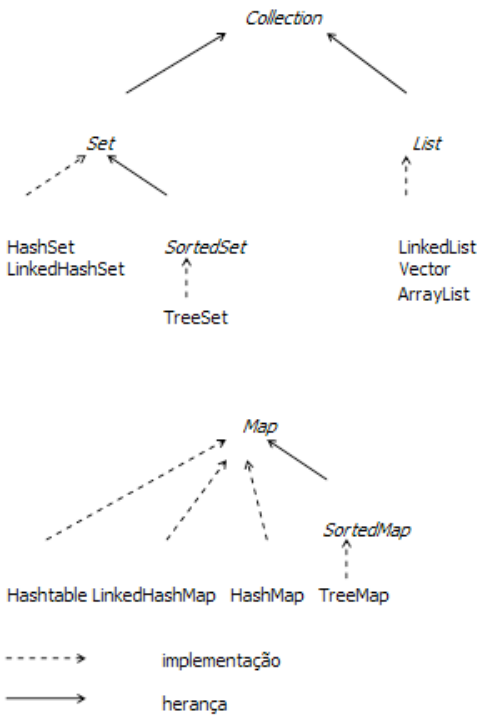

Figura 1. Hierarquia de interfaces/classes de coleção de dados em Java.

Observação: Os nomes identificados na figura acima em itálico são interfaces, os demais são classes.

8. Classes Internas

As classes internas são divididas em:

- Classe estática aninhada (top-level class)
- Classe interna comum
- Classe interna local de método
- Classe interna anônima

Classes estáticas

Top-level class.

Uma classe estática não tem acesso aos membros da instância da classe encapsulada, somente os membros estáticos têm a mesma visibilidade de uma classe externa.

Modificadores que podem ser atribuídos a uma classe interna estática:

- **static** - obrigatório é claro
- **protected**
- **public**
- **private**
- **abstract**
- **final**

Nunca usar os modificadores abstract e final simultaneamente!

```
public class Ex03 {
    static int CountStatic = 0;
    int CountNonStatic = 0;
    public static class Inner {
    }
}
```

Classe interna comum

NUNCA poderá ter uma instância de uma classe interna sem que haja uma instância de uma classe externa.

Qualquer tentativa de instanciação da classe Inner sem que haja um OBJETO do tipo Outer, não funciona!

```
public class Outer {
    public static void main(String[] args) {
        Outer o = new Outer();
        Inner i = o.new Inner();
    }

    class Inner {
    }
}
```

Classes internas e membros externos

Instanciando um objeto interno fora da classe externa

Referenciando a classe externa de dentro da classe interna

Modificadores aplicados as classes internas comuns:

- **final**
- **abstract**
- **public**
- **private**
- **protected**
- **static** - com uma exceção citada adiante.
- **Strictfp**

Classe interna local de método

```
public class Outer {
    public void see() {
        class Inner { }
    }
}
```

Uma classe de método tem o modificador **private** por padrão, visto que não pode ser instanciada de nenhum outro local senão o método que encapsula a classe.

Não tente instanciar a classe interna antes de criar externa!

Os únicos modificadores aplicados a essa classe são:

- **abstract**
- **final**

Nunca os dois ao mesmo tempo!!!

Variáveis:

Uma classe interna de método não pode referenciar as variáveis locais de método.

Porém, se a variável for **final** poderá ser referenciada.

Classes internas anônimas

Compilador só irá conhecer os métodos definidos na classe pai - qualquer tentativa de execução ou chamada de um método não existente, causará um erro de compilação: cannot resolve symbol.

```
class Empregado {
    public void trabalhar() {
       System.out.println("trabalhar");
    }
}

class QuadroFuncionario {
    Empregado mgr  = new Empregado() {
        public void trabalhar() {
            System.out.println("mandar");
        }

        public void demite() {
            System.out.println("demite");
        }
    };

    public void work() {
       mgr.trabalhar();
       mgr.demite();
    }
}
```

Só variáveis **final** podem ser referenciada.

Se precisar referenciar uma variável local dentro de uma classe anônima use array. Array é final seus elementos não.

Classes finais não podem ser anônimas: Nunca podemos ter uma classe anônima a partir de uma classe final, pois uma classe anônima nada mais é do que uma herança de uma outra classe onde a subclasse

9. Segmentos

Maneiras de implementar:

a) Estendendo da Classe **Thread**

b) Implementando a Interface **Runnable**

Classe herda de Thread: implementa seu método abstrato **run**, o código que se encontrar dentro é executado.

Classe implementa Runnable: implementa o método **run**, o método que é executado é o método da classe que implementa a interface Runnable.

A classe Thread implementa a interface Runnable.

Chamar o método start para que o escalonador saiba da existência da Thread, e deixe que o método run seja executado pelo escalonador. Se você não chamar o método start, sua thread nunca será executada.

Métodos importantes da classe Thread

- **run()** - é o código que a thread executará. Não gera nenhuma exceção.
- **start()** - sinaliza à JVM que a thread pode ser executada, mas saiba que essa execução não é garantida quando esse método é chamado, e isso pode depender da JVM.
- **isAlive()** - volta true se a thread está sendo executada e ainda não terminou.
- **sleep()** - suspende a execução da thread por um tempo determinado. Membro estático. Ele resulta que a execução corrente de uma thread “durma” por um temo específicado em milisegundos.
- **yield()** -torna o estado de execução da thread atual para **executável** para que thread com prioridades equivalentes possam ser processadas. Membro estático. Sua mudança no estado da transição não é garantida.
- **currentThread()** - é um método estático da classe Thread que volta qual a thread que está sendo executada.
- **getName()** - volta o nome da Thread, você pode especificar o nome de uma Thread com o método setName() ou na construção da mesma, pois existe os construtores sobrecarregados.
- **join() -** espera por uma thread específica morrer. Pode ser passado como parâmetro o tempo em mili e nano segundos.

Métodos herdados de java.lang.Object relacionados com Threads:

- public final void wait()

Resulta que uma única thread espere até que outra thread qualquer invoque o método **notify()** ou **notifyAll()** para esse objeto, ou até que um tempo específicado tenha decorrido.

- public final void notify()

Acorda uma única thread que está esperando seu controlador do objeto.

- public final void notifyAll()

  Acorda todas as threads que estão esperando seu controlador do objeto.

Esses métodos são definidos na classe Object, todos os demais objetos tem esses métodos, até mesmo a classe Thread. Eles SEMPRE devem ser chamados de um contexto sincronizados.

Não se esqueça: Nunca se reinicia uma *thread* já executada.

- Não existe *lock* para primitivos, então, eles não podem ser usados para sincronizar *threads*.

Observações sobre Thread:

- **join(), sleep(), wait()**: Gera a exceção InterruptedException. Pode ser especificado um parâmetro para timeout.
- Um erro de compilação ocorrerá se a expressão produzir qualquer valor de tipo primitivo.
- Se a execução de um bloco finaliza normalmente então o lock é liberado.
- Se a execução de um bloco finaliza bruscamente então o lock é liberado.
- Uma thread pode segurar mais de um lock ao mesmo tempo.
- Instruções sincronizadas podem ser aninhadas.
- Instruções sincronizadas com expressões iguais podem ser aninhadas.

Modelo de estados de transições do clico de vida de uma thread:

- Do estado pronto para rodando. (Ready → Running)
- Do estado rodando para o não rodado (Running → Not-Runnable)
- Do estado Rodado para o pronto (Running → Ready)
- Do estado não rodado para o estado pronto. (Not-Runnable → Ready)

Valores de definições para prioridades de execução das Threads:

- Thread.MAX_PRIORITY == 10
- Thread.NORM_PRIORITY == 5
- Thread.MIN_PRIORITY == 1

Invocar o método start() para uma thread que já foi iniciada gerará a exceção: **IllegalThreadStateException.**

Se o método start() é invocado para uma thread que já está em execução, então provavelmente a exceção será gerada: **IllegalThreadStateException**.

Se para thread já está morta, uma segunda tentativa de iniciá-la novamente for executada provavelmente a thread será ignorada e nenhuma exceção será gerada.

Para propósito de exame de certificação, a exceção é sempre gerada em resposta a uma segunda tentativa de iniciar uma thread.

É necessário conhecer os construtores da classe Thread se o intuito for a certificação.

Construtores para a classe Thread
Thread()
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target , String name)
Thread(ThreadGroup group, Runnable target , String name)
Thread(ThreadGroup group, String name)

Para Saber Mais

SIERRA, Kathy e BATES, Bert - Certificação Sun Para Programador & Desenvolvedor Java 2

http://www.atimonet.com.br/kuesley/estudojava/index.html

http://java.sun.com/reference/api/index.html

Simulados/Exercícios

http://www.javacertificate.com/

http://www.danchisholm.net/july21/comprehensive/index.html

http://www.javablackbelt.com/

http://www.guj.com.br/posts/list/13146.java

http://www.javaranch.com/mock.jsp