Plataforma Java

Java Fundamentos

Sumário

1. Linguagem Java
2. Java Virtual Machine
3. Modelo de Memória JVM
4. IDE’s
5. Tipos
6. Programas Standalone
7. Estruturas de Controle
8. Estruturas de Repetição
9. Debugando Aplicações
10. Arrays

Linguagem Java

1. História
2. Características da Linguagem
3. JRE
4. JDK
5. Plataforma Java
6. Compilação e Execução
7. JSE
8. JSE - HotSpot

Linguagem Java

História

Criador: James Gosling e Equipe em 1994/5

Empresa: SUN Microsystems

Objetivo: Criar software embarcados

Inicialmente tentavam utilizar o C/C++, mas logo perceberam que seria muito difícil utilizar uma linguagem que gerava código *nativo* para um padrão de software embarcado.

Crou-se então uma linguagem e uma VM

Linguagem Java

Características da Linguagem

1. Influência do C++
2. Linguagem muito rica e poderosa
    a. Orientada a Objeto
    b. Muitos Recursos Poderosos
        i. Annotation, Generics (Template), Reflection, Lambda, Streams, Multthread, Distribuída, Portável, Performática
    c. Muito Madura (muitas APIs e Frameworks)
    d. Recursos avançados para desenvolver vários tipos de aplicação
        i. Web, Standalone, API Rest, GUI, Mobile, etc
3. Compila o código fonte para *ByteCode*, um código intermediário capaz de ser interpretado pela JVM
4. Não possui aritmética de ponteiro (mais simples então)

Linguagem Java

Java Runtime Environment - JRE

1. Conjunto de ferramentas, interpretador de ByteCode e bibliotecas, necessários para a execução de programas desenvolvidos em Java
2. Pode ser instalado separadamente da JDK

Linguagem Java

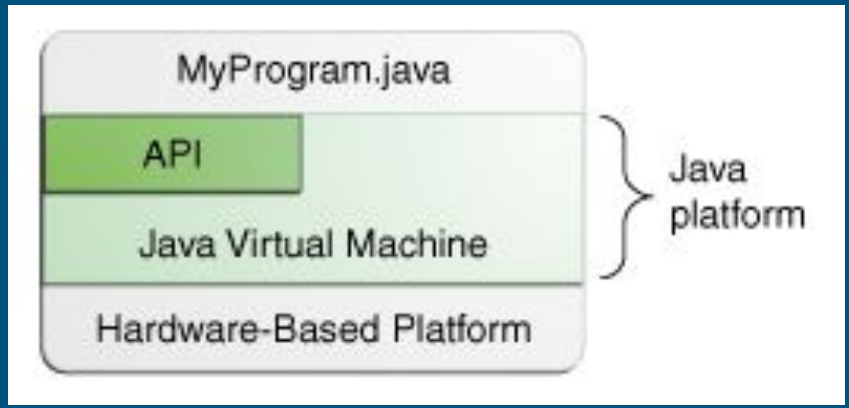
Java Development Kit - JDK

1. Conjunto de ferramentas, compilador Java, interpretador de ByteCode e bibliotecas, necessários para construir programas desenvolvidos na linguagem Java
2. O JDK já inclui o JRE
3. OpenJDK (mundo Linux)

Linguagem Java

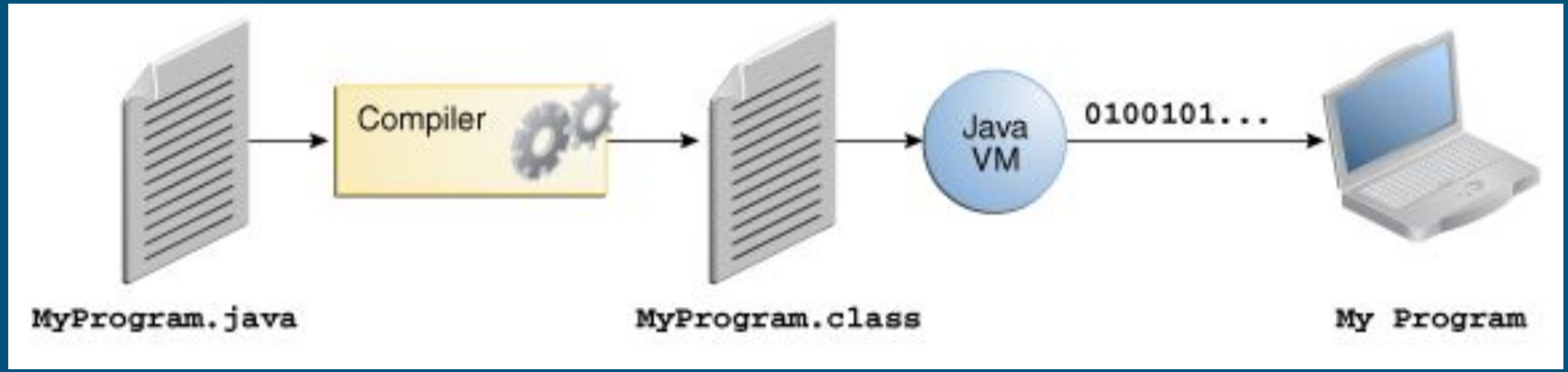
Plataforma Java

1. Plataforma: Software e Hardware em que um programa é executado
2. API: Conjunto extenso de utilitários e componentes para ser utilizado pelo programador para construir programas Java



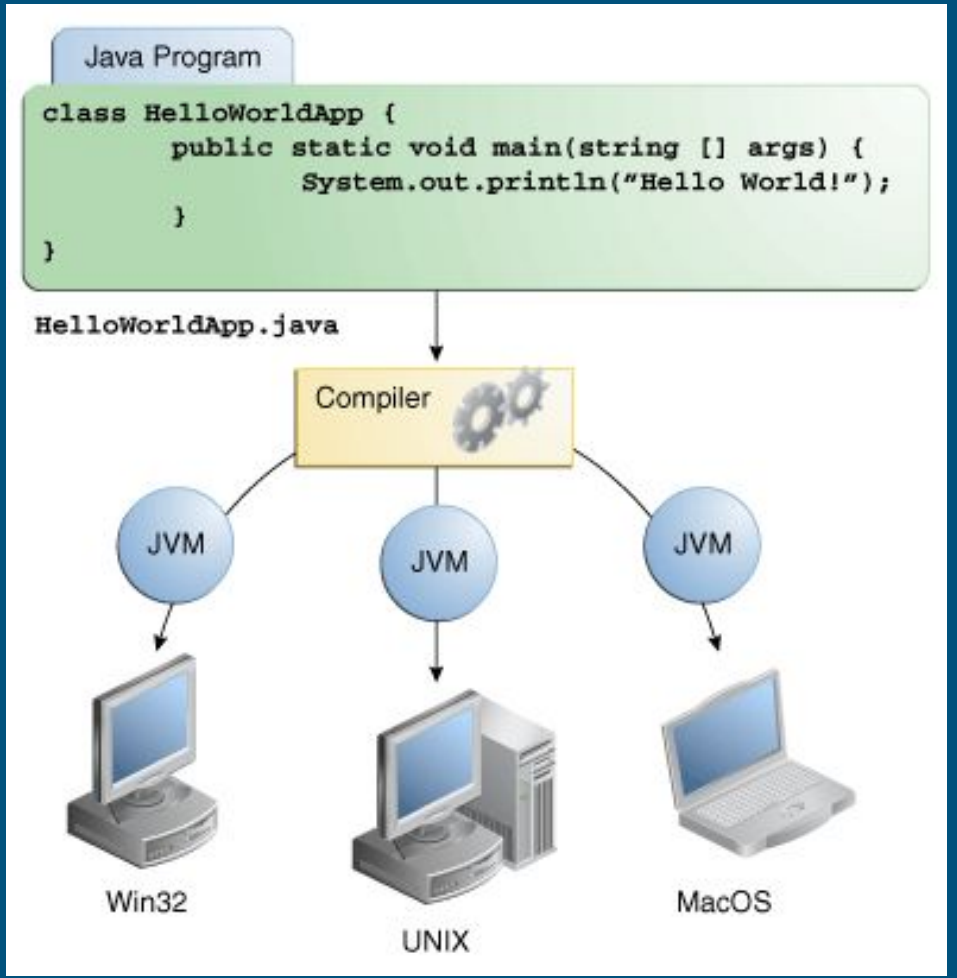
Linguagem Java

Compilação e Execução



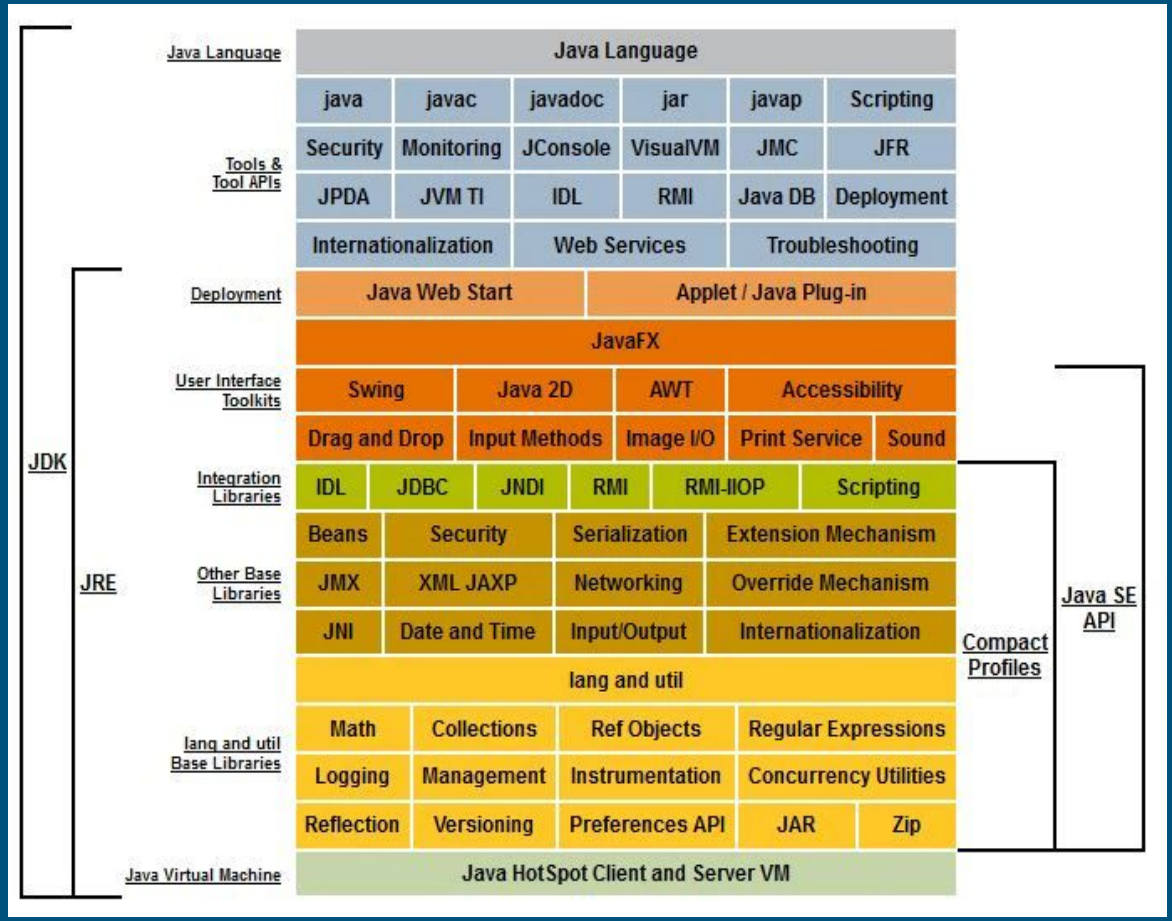
Linguagem Java

JVM e Portabilidade



Linguagem Java

Java SE - Visão Geral



Linguagem Java

Java SE HotSpot

1. Componente *“Core”* da JVM
2. Implementa a Especificação da JVM e é compartilhado como um *“Shared Library”* na JRE
3. Tratando-se de Engine de Execução possui diversos recursos
    a. Thread
    b. Sincronização de Objetos
4. Possui a característica de adaptar-se as configurações da plataforma em que está executando
    a. Selecionará o compilador
    b. Configuração do Heap
    c. Garbage Collector
5. OBJETIVO: Sempre procurar a melhor performance, por isso o Java possui uma performance excelente (considerando linguagens não nativas)

Java Virtual Machine

“Programa interpretador do byte code gerado pelo compilador Java”

“Write once, run anywhere”

JVM

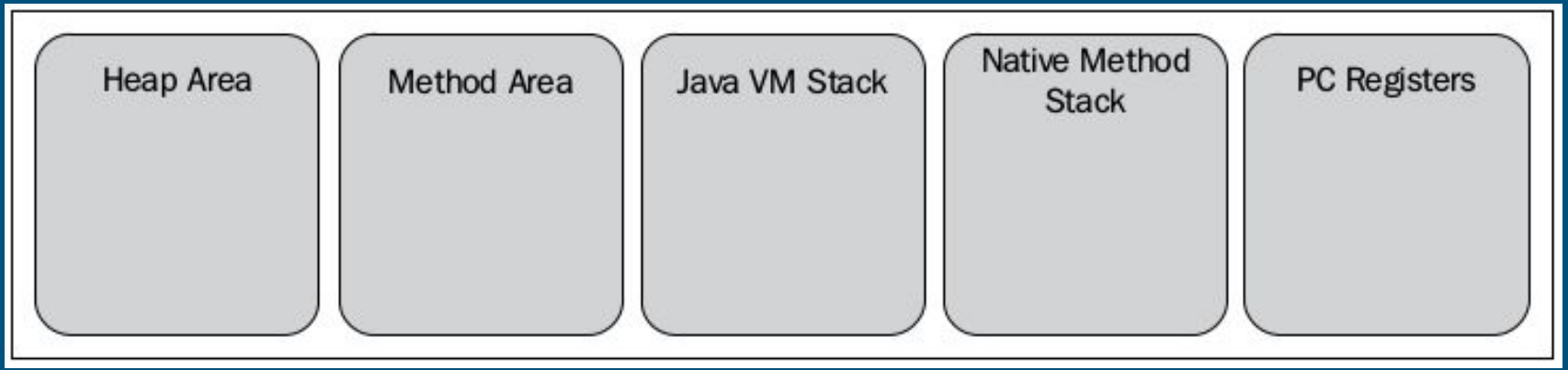
Implementações

1. Client VM: Tipicamente usado para aplicações cliente.
    a. Reduzir o tempo de inicialização
    b. Reduzir a utilização de memória
    c. Para invocar este modo utilize na linha de comando a opção: -client
2. Server VM
    a. Maximizado para velocidade de execução
    b. Para invocar este modo utilize na linha de comando a opção: -server

Modelo de Memória

Modelo de Memória

Áreas de Memória da JVM



Modelo de Memória

Heap

1. Área de memória de runtime, na qual a memória é *“alocada”* para criação das instâncias e arrays
2. Criada durante a inicialização da aplicação
3. Esta área é solicitada pelo gerenciador de memória (Garbage Collector)
4. Pode ser fixa ou dinâmica
5. Não necessita ser alocada *“continuamente”*

Modelo de Memória

Method Area

1. Área de memória para armazenar o código do programa, normalmente chamada de *“text segment”*
2. Armazena estruturas necessárias para cada classe ou interface tal como o *“constant pool”* e informações sobre atributos (fields) e métodos (methods), além do código para métodos e construtores, incluindo-se métodos especiais usados em classes, instâncias e inicializações de interfaces
3. Criada na inicialização da VM

Modelo de Memória

JVM Stack

1. Cada thread tem uma stack criada no momento da criação da thread
2. Mantém variáveis locais e resultados parciais para gerenciamento de chamadas de métodos
3. Pode ter tamanho fixo ou dinâmico

Modelo de Memória

Native Method Stack (C stacks)

1. Stack especial para métodos nativos, ou seja, métodos desenvolvidos em linguagens nativas, em especial C
2. Tipicamente alocado por thread
3. Pode ter tamanho fixo ou dinâmico

Modelo de Memória

PC Registers

1. Cada thread tem seu próprio PC (program counter) register
2. Em cada ponto, cada thread está executando o código de um determinado método, nomeado *“current method”* para a thread em questão
3. Como uma aplicação Java pode conter código nativo, se o método em execução não é nativo o PC aponta para a instrução corrente na JVM, caso contrário, segundo a especificação o valor é indefinido (undefined)

Modelo de Memória

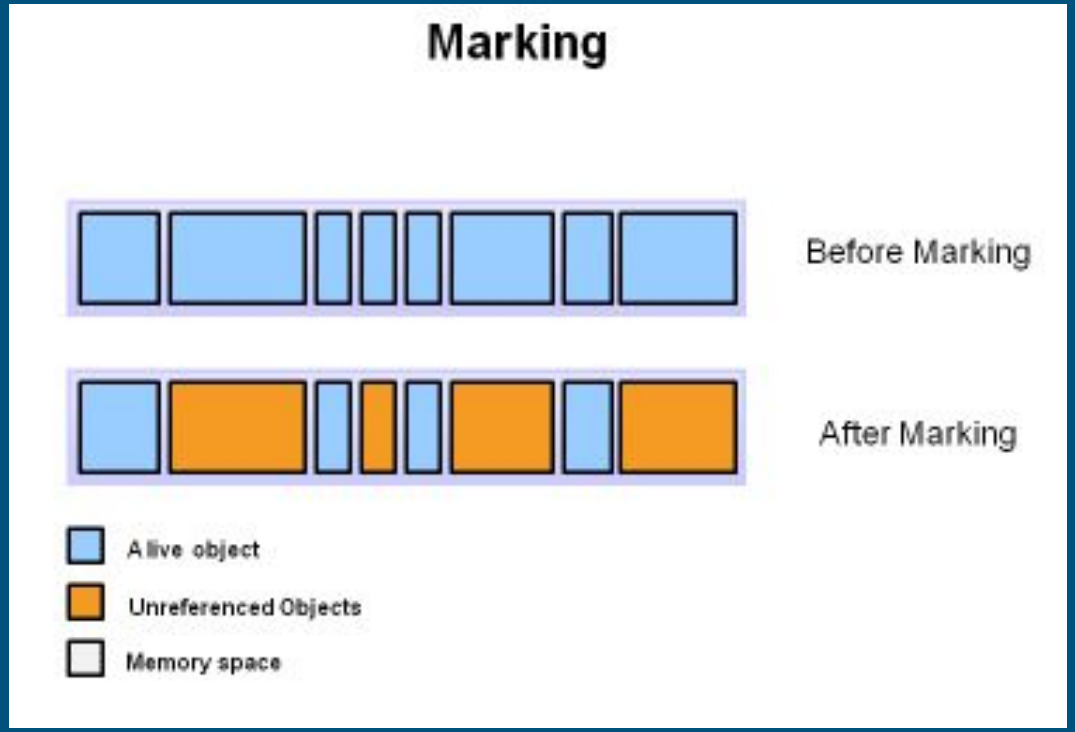
Garbage Collector

1. Thread responsável pela liberação de memória ocupados por objetos não mais “referenciados” pelo programa
2. Algoritmos de Garbage Collect
3. Elemento importante para a performance das aplicações
4. Estrutura de Memória Generacional
    a. A vida dos objetos é organizado em gerações
    b. Yung Generation
        i. Coleta de memória mais frequentes, eficientes e rápidas
        ii. Normalmente possui objetos menores e comumente possui muitos objetos “não mais referenciados“
    c. Old Generation
        i. Objetos que sobrevivem a coleta na young generation eventualmente são promovidos para a old generation

Modelo de Memória

GC - Marking

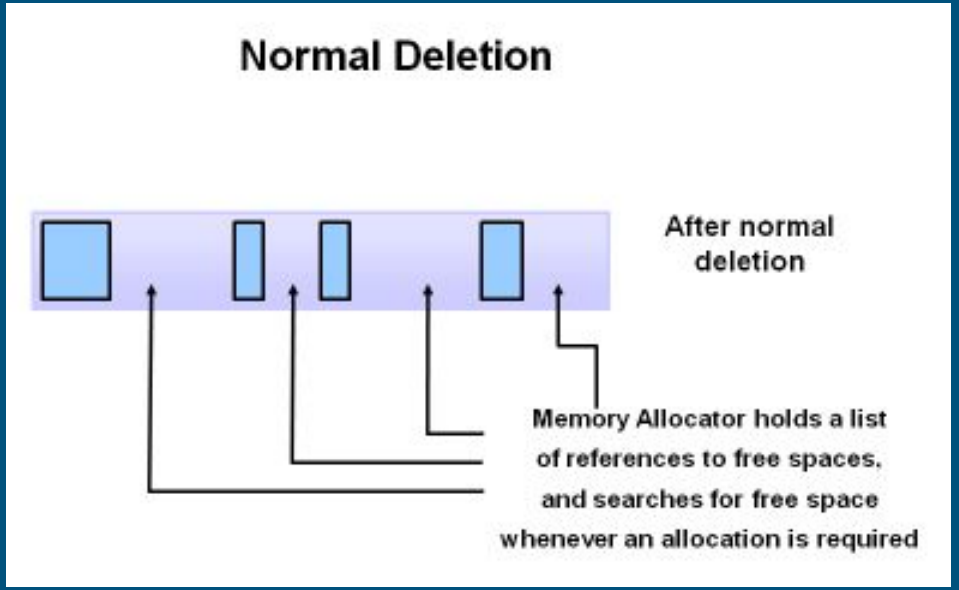
1. Fase em que o GC determina quais objetos são referenciados e quais não são
2. Todos os objetos são *scaneados*
3. Pode ser um processo com custo elevado



Modelo de Memória

GC - Normal Deletion

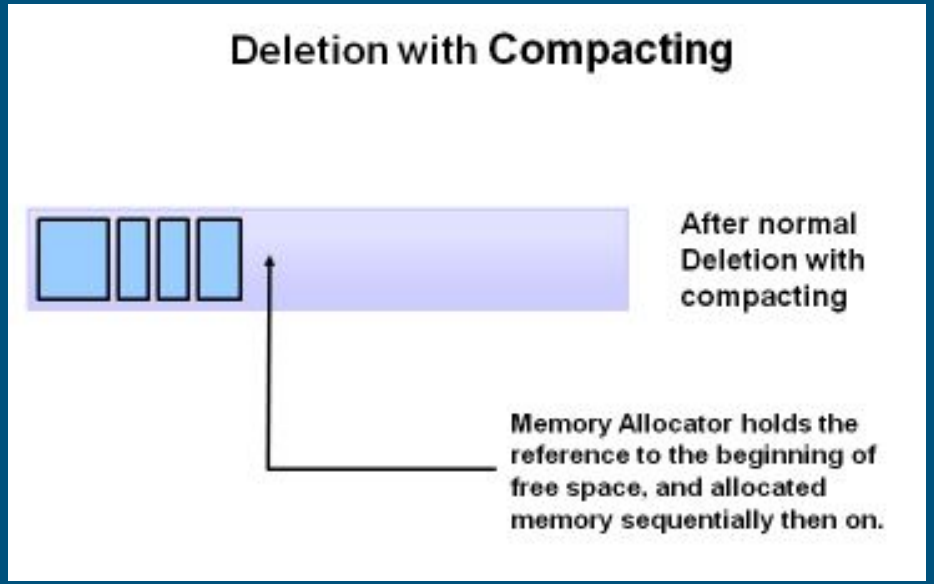
1. Remove objetos não referenciados liberando a área de memória
2. MemoryAllocator mantém uma lista das referências livres para utilizar na alocação quando necessário



Modelo de Memória

GC - Deletion with Compacting

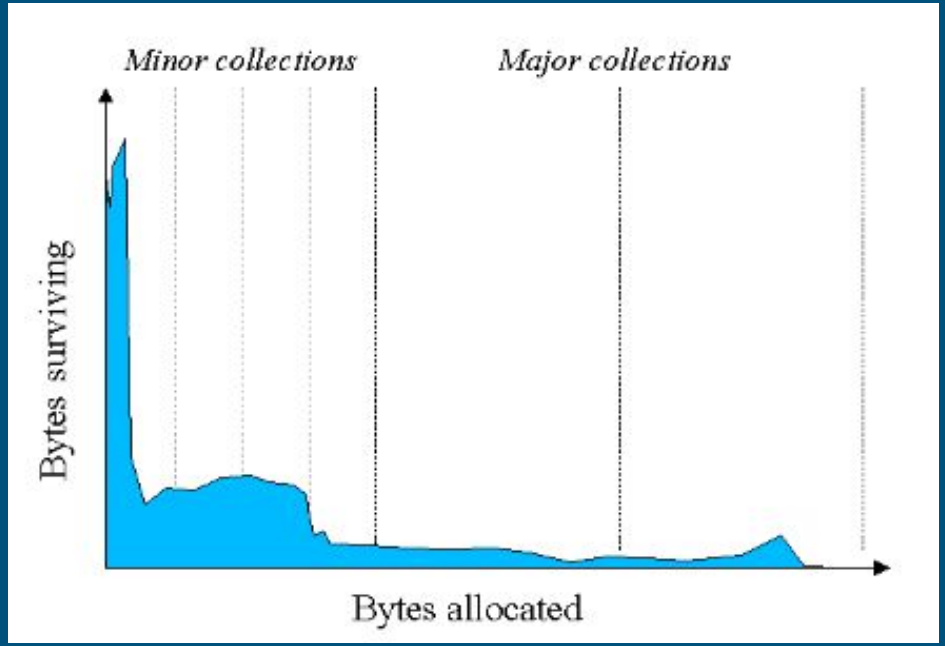
1. Após a remoção, para melhorar a performance do processo de alocação de memória, os objetos referenciados podem ser compactados, movendo-os e tornando-os contíguos
2. MemoryAllocator mantém um ponteiro para o início da área livre



Modelo de Memória

Por que uma estrutura orientada a Gerações?

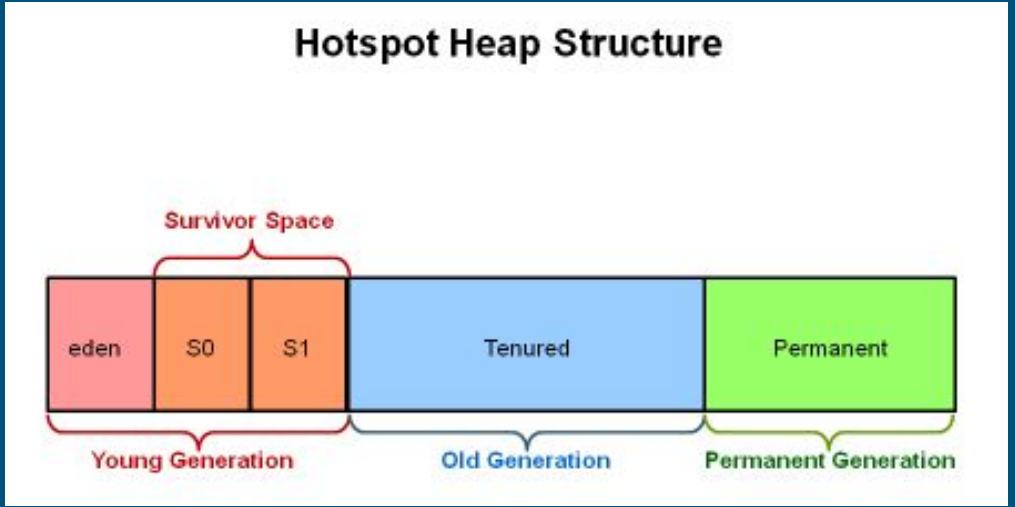
1. O processo de marcar e compactar todos os objetos na JVM é naturalmente *“ineficiente”*
2. Normalmente o tempo de vida de um objeto, estatisticamente, é pequeno



Modelo de Memória

GC Generations

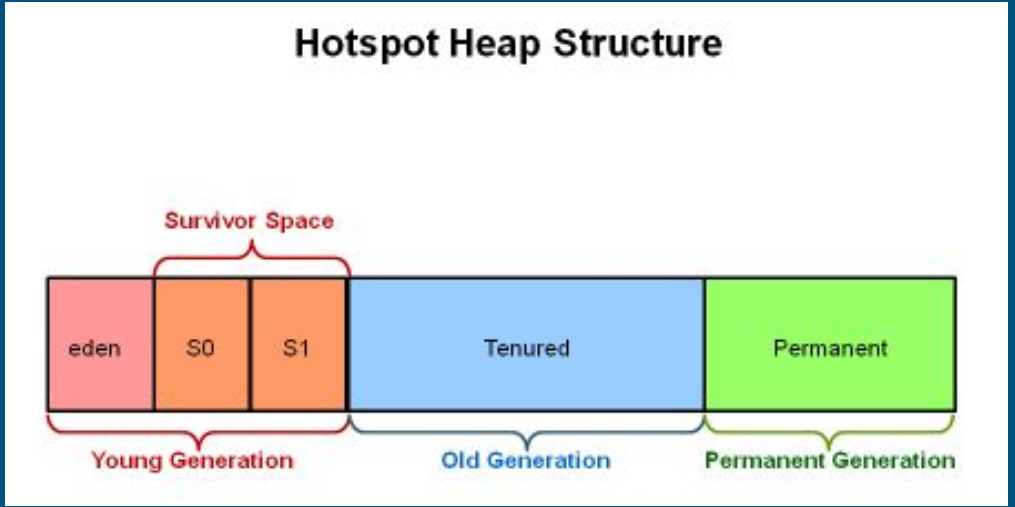
1. Young Generation: Onde todos os novos objetos são alocados. Quando ela é preenchida ocove um *“Minor garbage Colection”*
2. Minor Collections podem ser otimizados assumindo uma alta taxa de mortalidade.
3. Uma Young Generation cheia de objetos mortos é coletada rapidamente.
4. Objetos sobreviventes são eventualmente movidos para a Old Generation



Modelo de Memória

GC Generations

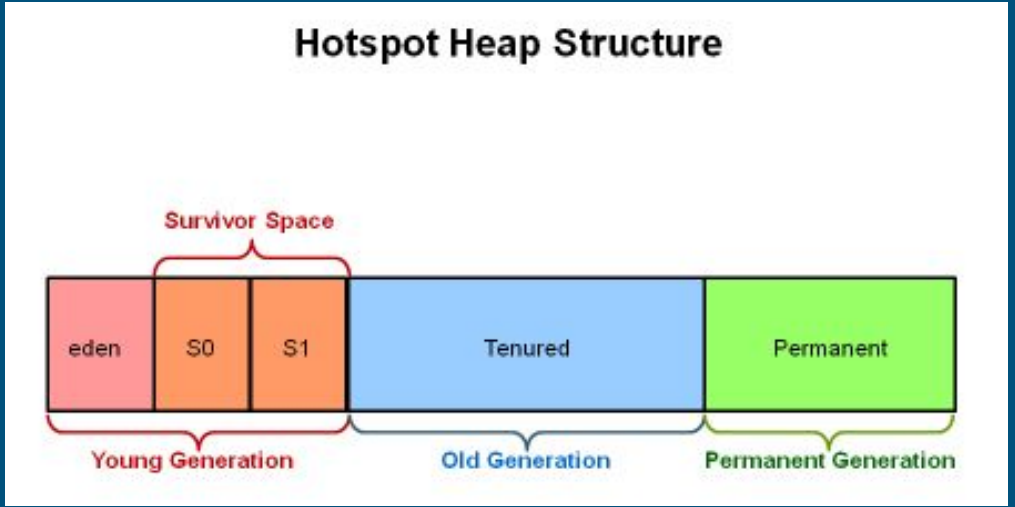
1. Stop The World Event: Todas as *Minor Collections* são eventos do tipo *"Stop The World"*, ou seja, todas as threads da aplicação deverão parar para que a thread do GC execute e realize seu trabalho de coleta de objetos não referenciados



Modelo de Memória

GC Generations

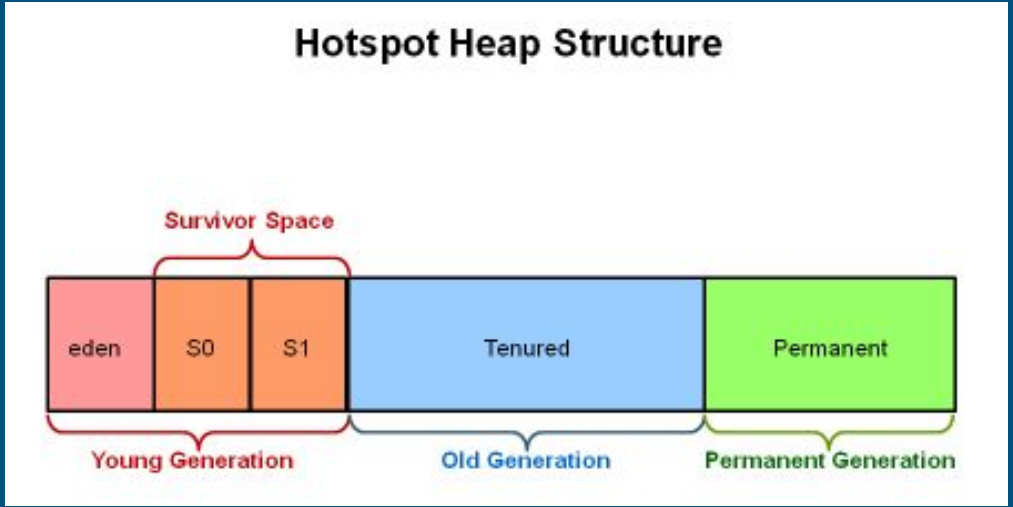
1. Old Generation: É usada para armazenar objetos que possuem um ciclo de vida mais longo, ou seja, sobrevivem a ponto de serem transferido para a Old Generation
2. Normalmente existe um limite na quantidade de sobrevivência de Minor Collection Events, para que a transferência para o Old Generation
3. Quando a Old Generation precisa ser coletada ocorre um *Major Garbage Collection*



Modelo de Memória

GC Generations

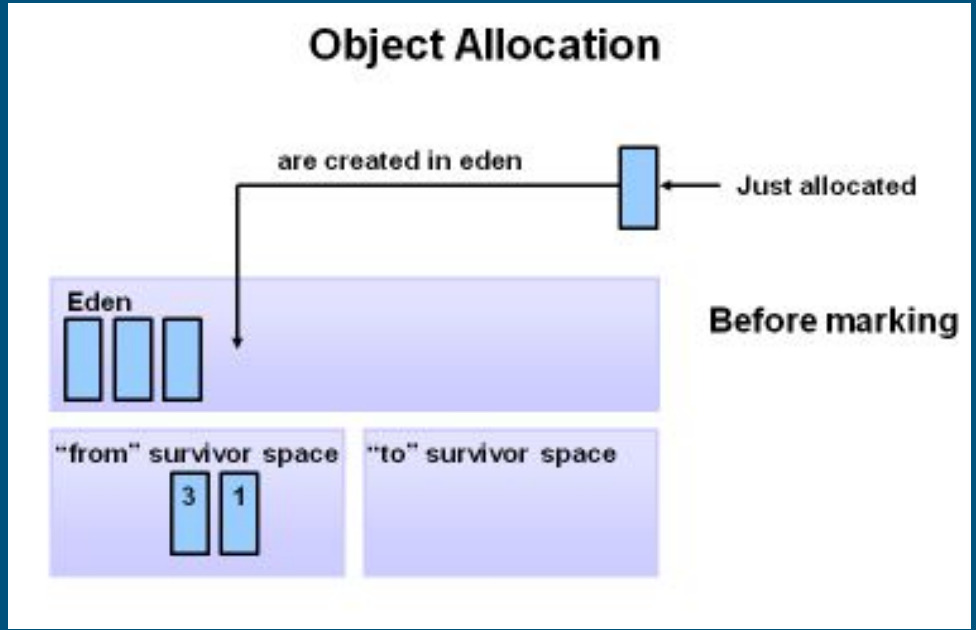
1. Normalmente *Major Garbage Collection* é mais lenta porque envolve todos os objetos sobreviventes, portanto para melhorar a responsividade da aplicação um Major GC precisa ser evitado ou minimizado
2. O tamanho do evento Stop The World para um Major GC é afetado pelo tipo de GC utilizado na Old Generation
3. Permanent Generation contem metadados requeridos pela JVM para descrever das classes e métodos usados pela aplicação. A JVM pode coletar dados na PG se os objetos não são mais necessários e precisar de memória



Modelo de Memória

Processo de Alocação

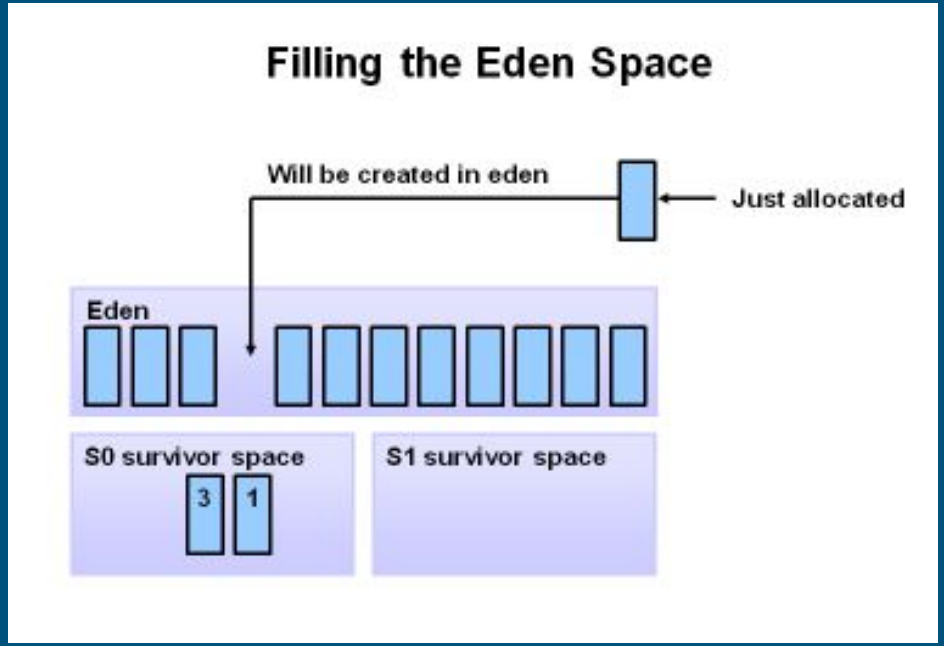
1. Qualquer objeto recém alocado é mantido no Eden
2. Ambas as áreas de *survivor* iniciam vazias



Modelo de Memória

Processo de Alocação

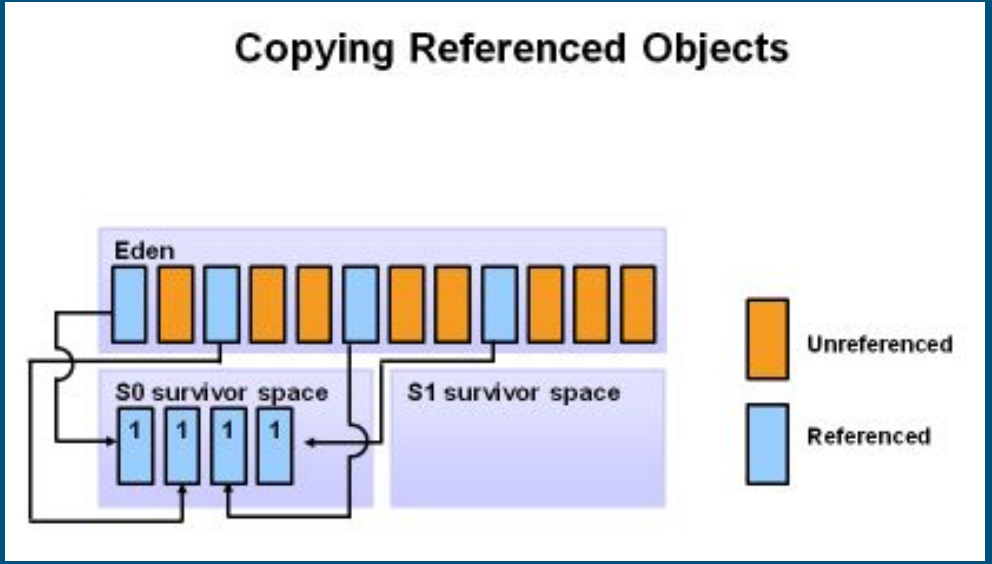
1. Quando o Eden é completamente ocupado, um Minor GC é disparado



Modelo de Memória

Processo de Alocação

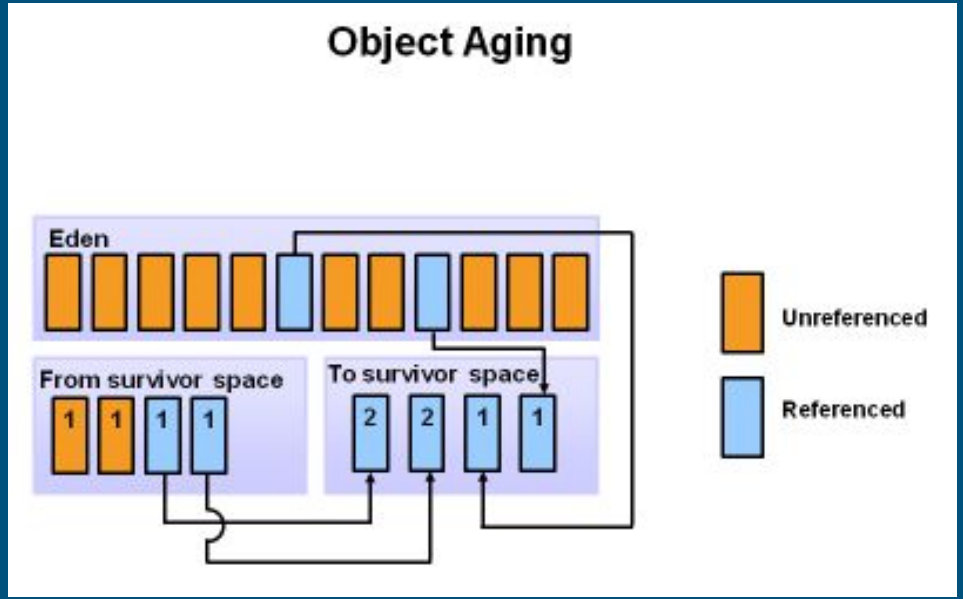
1. Objetos referenciados são movidos para a primeira área survivor
2. Objetos não referenciados são removidos do Eden



Modelo de Memória

Processo de Alocação

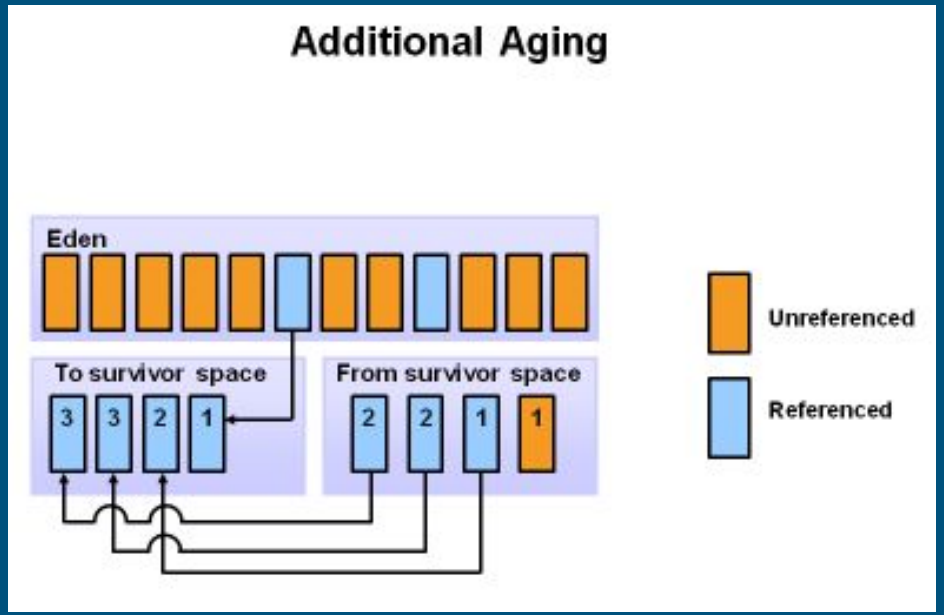
1. No proximo Minor GC o mesmo acontece, objetos não referenciados são removidos do Eden ou S0, os sobreviventes são movidos para S1
2. Objetos que foram movidos de S0 para S1 tem sua idade incrementada



Modelo de Memória

Processo de Alocação

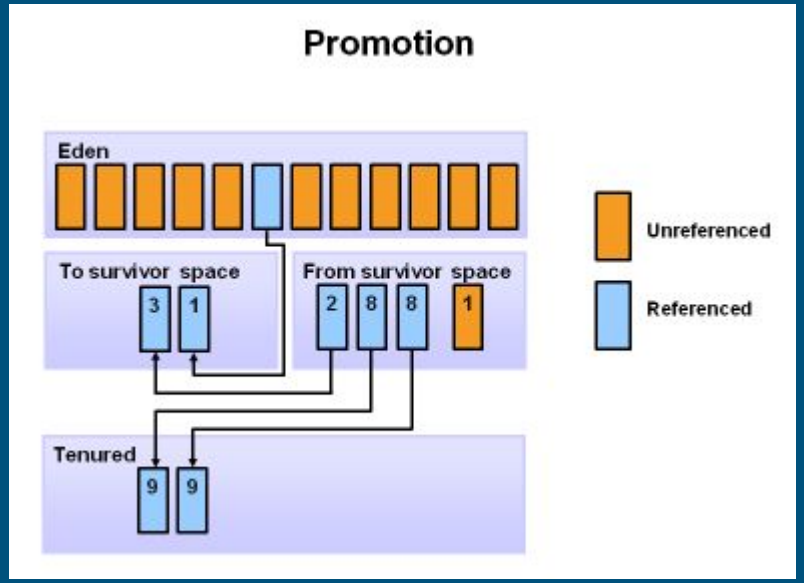
1. No proximo Minor GC o mesmo acontece, só que objetos sendo movidos de S1 para S0

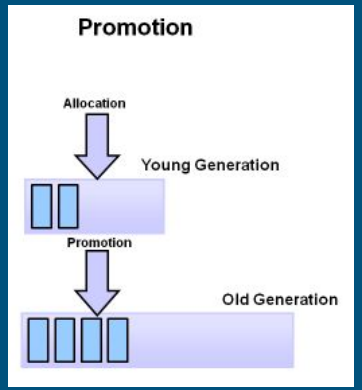


Modelo de Memória

Processo de Alocação

1. Promoção é quando um objeto sai da Young Generation (S0/S1) e é movido para Old Generation (Ternured)
2. Isto ocorre quando a “idade” do objeto atinge determinado limite

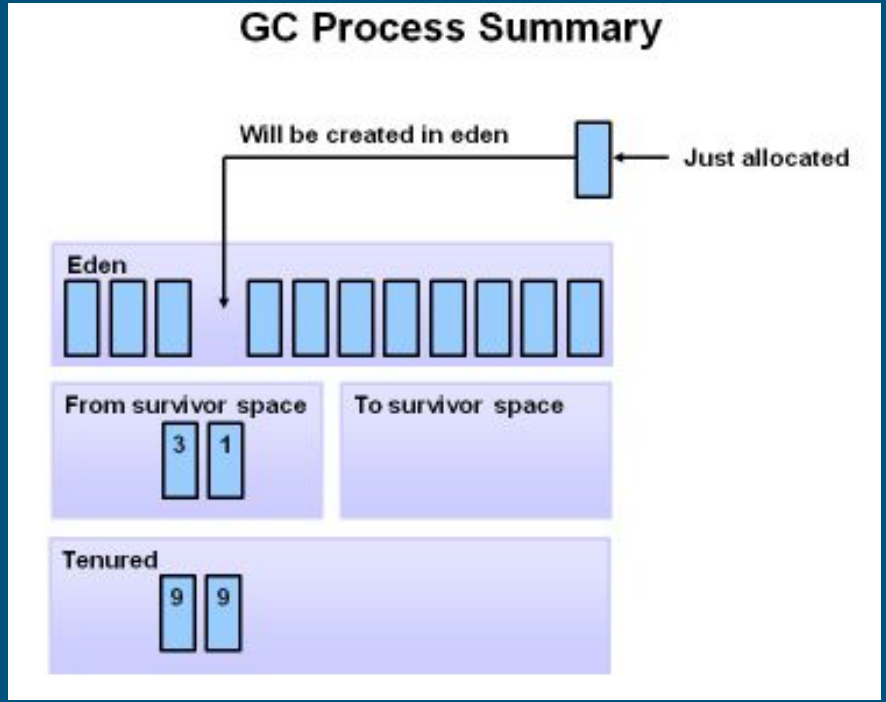




Modelo de Memória

Configuração Memória

1. Eventualmente uma Major GC ocorre para liberar memória na Old Generation



Modelo de Memória

Processo de Alocação

Switch	Description
-Xms	Sets the initial heap size for when the JVM starts.
-Xmx	Sets the maximum heap size.
-Xmn	Sets the size of the Young Generation.
-XX:PermSize	Sets the starting size of the Permanent Generation.
-XX:MaxPermSize	Sets the maximum size of the Permanent Generation

Modelo de Memória

Garbage Collector - Serial GC

1. Aplicações que não possuem longas pausas e rodam em máquinas do cliente
2. Único processador virtual para processar o trabalho de GC
3. Ambientes em que existam várias JVM executando, onde é melhor
4. Dispositivos com pouco poder computacional (poucos cores e pouca memória)
5. Habilitar Serial GC: -XX:+UseSerialGC

Here is a sample command line for starting the Java2Demo:

```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar
c:\javademos\demo\jfc\Java2D\Java2demo.jar
```

Modelo de Memória

Garbage Collector - Parallel GC

1. Utiliza múltiplas threads para a Young Generation Collection
2. Por padrão, se um host possui N CPUs, então será utilizado N threads na coleta
3. A quantidade de threads pode ser controlada: -XX:ParallelGCThreads=<desired number>
4. UseParallelOldGC

UseParallelGC

Multi-thread Young Generation Collection, Single Thread Old Generation Collection, Single Threaded Compaction Old Generation

UseParallelOldGC

Multi-thread Young Generation Collection, Multi Thread Old Generation Collection, Multi Threaded Compaction Old Generation

Modelo de Memória

Garbage Collector - Concurrent Mark Sweep

1. Tenta minimizar as pausas realizando a coleta paralelamente as threads da aplicação
2. Normalmente não faz compactação
3. Se a fragmentação tornar-se um problema então deve-se alocar mais memória
4. Para Young Generation utiliza o mesmo algoritmo do Parallel GC

Utilização

Aplicações que requerem baixo tempo de pausa e podem compartilhar recursos com o GC, exemplo?

- Aplicações Desktop que respondem a eventos
- Servidor Web respondendo a requisições
- Banco de dados respondendo a consultas

Modelo de Memória

Garbage Collector - Concurrent Mark Sweep

To enable the CMS Collector use:
`-XX:+UseConcMarkSweepGC`
and to set the number of threads use:
`-XX:ParallelCMSThreads=<n>`

Here is a sample command line for starting the Java2Demo:

```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseConcMarkSweepGC -
XX:ParallelCMSThreads=2 -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar
```

Modelo de Memória

Garbage Collector - G1

1. Atualmente é o mais indicado
2. Surgiu com o Java7
3. Substitui o CMS GC
4. Paralelo, Concorrente
5. Compactação incremental

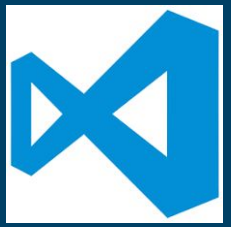
To enable the G1 Collector use:

```
-XX:+UseG1GC
```

Here is a sample command line for starting the Java2Demo:

```
java -Xmx12m -Xms3m -XX:+UseG1GC -jar c:\javademos\demo\jfc\Java2D\Java2demo.jar
```

IDE



“Integrated Development Environment”

Eclipse, Netbeans VSCode, IDEA

Tipos

1. Tipos Primitivos
    a. int
    b. short
    c. byte
    d. char
    e. float
    f. double
2. Tipos de Referência (*“pointers”*)
    a. Object
    b. String
    c. User Custom Types

Tipos

Tipos Primitivos

1. byte:
    a. 8 bit signed
    b. -128 a 127
    c. Economizar memória em grandes arrays
    d. Substituir int
2. short:
    a. 16 bit signed
    b. -32768 a 32767
    c. Economizar memória em grandes arrays
    d. Substituir int

Tipos

Tipos Primitivos

1. int:
   a. 32 bit signed
   b. -2^{31} a $2^{31}-1$
2. long:
   a. 64 bit signed
   b. -2^{63} a $2^{63}-1$
3. float
   a. 32 bit IEEE 754
   b. Não deve usado para alta precisão
4. double
   a. 64 bit IEEE 754
   b. Não deve usado para alta precisão

Tipos

Tipos Primitivos

1. boolean:
    a. true/false
2. char:
    a. 16 bit Unicode
    b. ‘\u0000’ até ‘\uffff’ (65535)

Tipos

Tipos Primitivos

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Tipos

Tipos Primitivos - Literais

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

```
// The number 26, in decimal
int decVal = 26;
//  The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

Tipos

Underscore in Literais Numéricas

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_____2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

Tipos

Tipos de Referência

1. Diz respeito a variáveis do tipo de Objeto
2. Exemplos
    a. Object
    b. String
    c. Tipos customizados pelo Programados
3. *Ponteiros* (sem a aritmetica de ponteiro)
4. null (referência nula, não *"aponta"* para nenhum objeto criado)

Programas Standalone

Programas que executam “fora” de qualquer estrutura tal como um container (web, ejb).

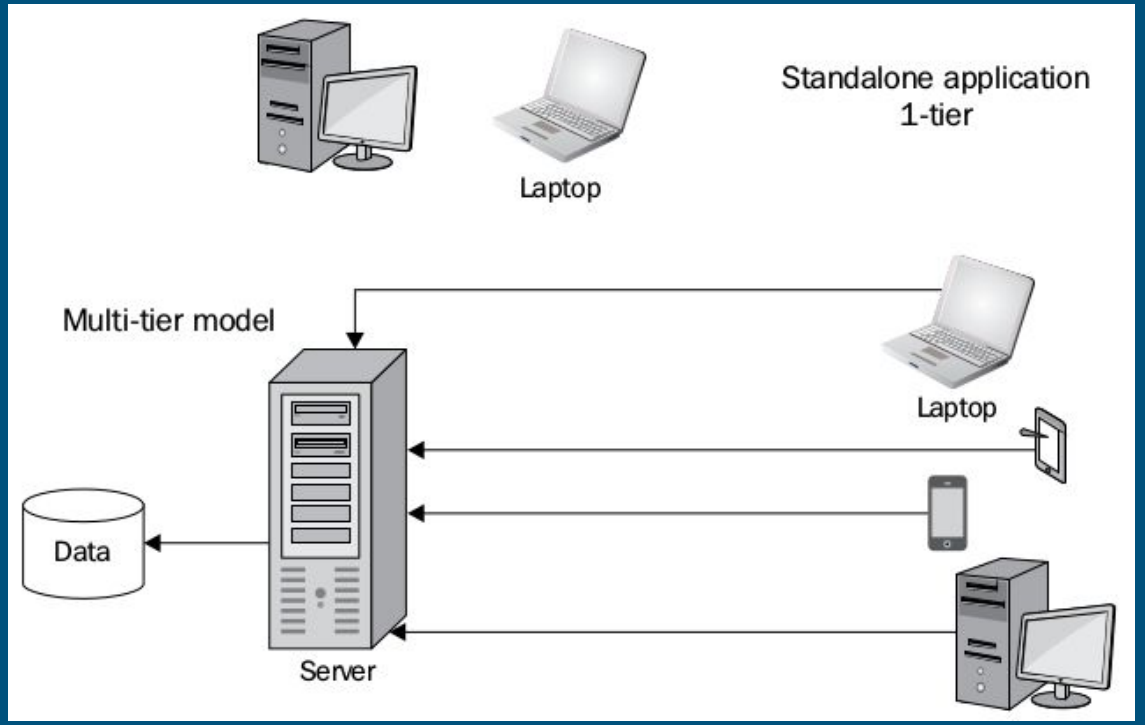
São “executáveis” simples, que iniciam a partir do método *“main”*

Executam num computador e podem acessar um servidor

Programas Standalone

Arquitetura

- Executa em uma única máquina
- Conecta com banco de dados local
- Projetado para o modelo de um único usuário concorrente
- Processamento local



Programas Standalone

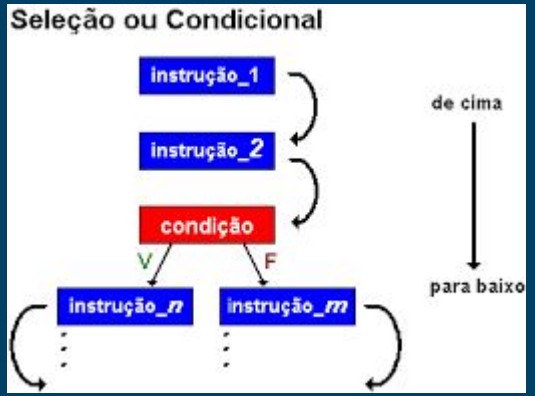
Leitura do Teclado

1. System.console()
2. Compilação
   a. javac console/Main.java
3. Execução
   a. java console.Main

```
String username;

out.print("Usuario: ");
username = System.console().readLine();

out.print("Senha: ");
char[] password = System.console().readPassword();

out.printf("Usuario: %s, Senha: %s\n\n", username, new String(password));
```

Estrutura de Controles

If, if else, if else if, switch



Estruturas de Controle

If, if else, if else if, switch

```
if (<cond>)
    comando;
```

```
if (<cond>) {
    comando1;
    comando2;
}
```

```
if (<cond>)
    comando;
else
    comando;
```

Estruturas de Controle

If, if else, if else if, switch

```
if (<cond>) {
    comando1;
    comando2;
} else {
    comando3;
    comando4;
}
```

```
if (<cond1>) {
    comando1;
} else if (<cond2>) {
    comando2;
} else {
    comando3;
}
```

Estruturas de Controle

If, if else, if else if, switch

```
switch (<var>) {
      case 1:
            comando1;
            break;
      case 2:
            comando2;
            break;
      default:
            Comando3;
}
```

Estruturas de Controle

If, if else, if else if, switch

```
int month = 8;
String monthString;
switch (month) {
    case 1:  monthString = "January";
             break;
    case 2:  monthString = "February";
             break;
    case 3:  monthString = "March";
             break;
    case 4:  monthString = "April";
             break;
    case 5:  monthString = "May";
             break;
    case 6:  monthString = "June";
             break;
    case 7:  monthString = "July";
             break;
    case 8:  monthString = "August";
             break;
    case 9:  monthString = "September";
             break;
    case 10: monthString = "October";
             break;
    case 11: monthString = "November";
             break;
    case 12: monthString = "December";
             break;
    default: monthString = "Invalid month";
             break;
}
```

Estruturas de Controle

If, if else, if else if, switch

```
int month = 8;

switch (month) {
    case 1:  futureMonths.add("January");
    case 2:  futureMonths.add("February");
    case 3:  futureMonths.add("March");
    case 4:  futureMonths.add("April");
    case 5:  futureMonths.add("May");
    case 6:  futureMonths.add("June");
    case 7:  futureMonths.add("July");
    case 8:  futureMonths.add("August");
    case 9:  futureMonths.add("September");
    case 10: futureMonths.add("October");
    case 11: futureMonths.add("November");
    case 12: futureMonths.add("December");
             break;
    default: break;
}
```

Estruturas de Controle

If, if else, if else if, switch

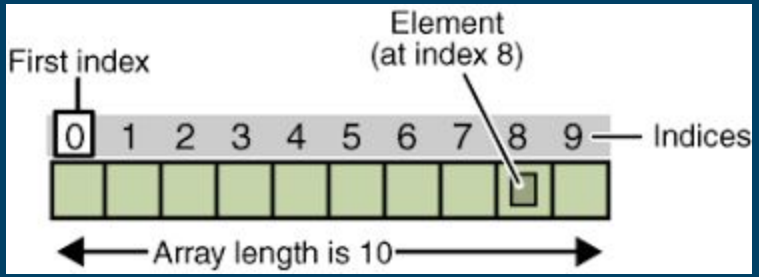
```
int month = 2;
int year = 2000;
int numDays = 0;

switch (month) {
    case 1: case 3: case 5:
    case 7: case 8: case 10:
    case 12:
        numDays = 31;
        break;
    case 4: case 6:
    case 9: case 11:
        numDays = 30;
        break;
    case 2:
        if (((year % 4 == 0) &&
             !(year % 100 == 0))
             || (year % 400 == 0))
            numDays = 29;
        else
            numDays = 28;
        break;
    default:
        System.out.println("Invalid month.");
        break;
}
```

Estruturas de Controle

If, if else, if else if, switch

```
switch (month.toLowerCase()) {
    case "january":
        monthNumber = 1;
        break;
    case "february":
        monthNumber = 2;
        break;
    case "march":
        monthNumber = 3;
        break;
    case "april":
        monthNumber = 4;
        break;
    case "may":
        monthNumber = 5;
        break;
    case "june":
        monthNumber = 6;
        break;
    case "july":
        monthNumber = 7;
        break;
    case "august":
        monthNumber = 8;
        break;
    case "september":
        monthNumber = 9;
        break;
    case "october":
        monthNumber = 10;
        break;
    case "november":
        monthNumber = 11;
        break;
    case "december":
        monthNumber = 12;
        break;
    default:
        monthNumber = 0;
        break;
}
```

Arrays


First index
Element
(at index 8)
0 1 2 3 4 5 6 7 8 9
Indices
Array length is 10

Arrays

Definição

1. Container de objetos (valores primitivos)
2. Acesso indexado
3. Tamanho fixo

```
// declares an array of integers
int[] anArray;

// allocates memory for 10 integers
anArray = new int[10];

// initialize first element
anArray[0] = 100;
// initialize second element
anArray[1] = 200;
// and so forth
anArray[2] = 300;
anArray[3] = 400;
anArray[4] = 500;
anArray[5] = 600;
anArray[6] = 700;
anArray[7] = 800;
anArray[8] = 900;
anArray[9] = 1000;
```

```
String[][] names = {
    {"Mr. ", "Mrs. ", "Ms. "},
    {"Smith", "Jones"}
};
// Mr. Smith
System.out.println(names[0][0] + names[1][0]);
// Ms. Jones
System.out.println(names[0][2] + names[1][1]);
```

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Arrays

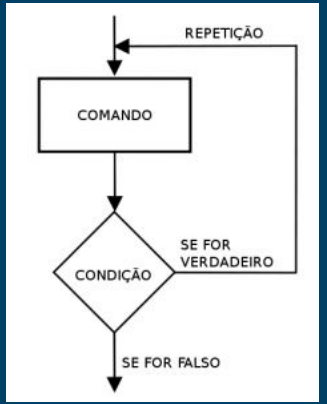
Cópia Eficiente

```
char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                    'i', 'n', 'a', 't', 'e', 'd' };
char[] copyTo = new char[7];

System.arraycopy(copyFrom, 2, copyTo, 0, 7);
System.out.println(new String(copyTo));
```

```
char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',
    'i', 'n', 'a', 't', 'e', 'd'};

char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

System.out.println(new String(copyTo));
```

Estrutura de Repetição

for, foreach, while, do while



Estrutura de Repetição

while, for, do..while, foreach

```
while (<cond>) {

}
```

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

}
```

```
do {

} while (<cond>)
```

```
int[] array = { … };

for (int x: array) {

}
```

Operadores

Operators	Precedence
postfix	*expr++ expr--*
unary	*++expr --expr +expr -expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Operadores

Aritméticos

Operator	Description
+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

Operadores

Unários

Operator	Description
+	Unary plus operator; indicates positive value (numbers are positive without this, however)
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
- -	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

Operadores

Igualidade e Relacionais

```
==      equal to
!=      not equal to
>       greater than
>=      greater than or equal to
<       less than
<=      less than or equal to
```

Operadores

Condicionais, ternário

```
&& Conditional-AND
|| Conditional-OR
```

```
int value1 = 1;
int value2 = 2;
if((value1 == 1) && (value2 == 2))
    System.out.println("value1 is 1 AND value2 is 2");
if((value1 == 1) || (value2 == 1))
    System.out.println("value1 is 1 OR value2 is 1");
```

```
int value1 = 1;
int value2 = 2;
int result;
boolean someCondition = true;
result = someCondition ? value1 : value2;
```

Operadores

instanceof

```
class Parent {}
class Child extends Parent implements MyInterface {}
interface MyInterface {}
```

```
Parent obj1 = new Parent();
Parent obj2 = new Child();

System.out.println("obj1 instanceof Parent: "
    + (obj1 instanceof Parent));
System.out.println("obj1 instanceof Child: "
    + (obj1 instanceof Child));
System.out.println("obj1 instanceof MyInterface: "
    + (obj1 instanceof MyInterface));
System.out.println("obj2 instanceof Parent: "
    + (obj2 instanceof Parent));
System.out.println("obj2 instanceof Child: "
    + (obj2 instanceof Child));
System.out.println("obj2 instanceof MyInterface: "
    + (obj2 instanceof MyInterface));
```

Operadores

Bitwise e Bit Shift

1. & => Operador AND
2. | => Operador OR
3. ^ => Operador XOR

```
int bitmask = 0x000F;
int val = 0x2222;
// prints "2"
System.out.println(val & bitmask);
```

Estrutura da Linguagem

Expressions, Statements and Blocks

1. Expressão: Construção feita com variáveis, operadores e invocação de métodos, respeitando a gramática da linguagem

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```

Estrutura da Linguagem

Expressions, Statements and Blocks

1. Comandos: Equivalente a sentenças em uma linguagem natural. Forma uma completa unidade de execução

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

```
// declaration statement
double aValue = 8933.234;
```

Estrutura da Linguagem

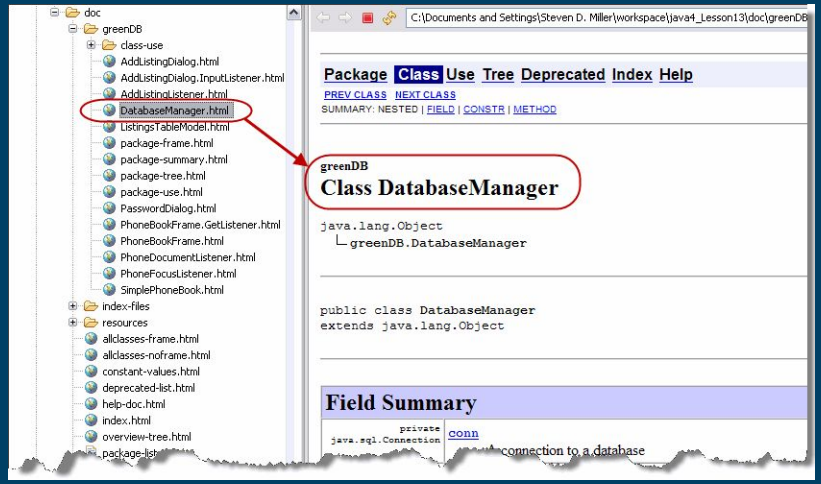
Expressions, Statements and Blocks

1. Block: Grupo de 1 (um) ou mais Statements

```
boolean condition = true;
if (condition) { // begin block 1
     System.out.println("Condition is true.");
} // end block one
else { // begin block 2
     System.out.println("Condition is false.");
} // end block 2
```

Debugando Aplicações Java

JavaDoc


doc
greenDB
class-use
AddListingDialog.html
AddListingDialog.InputListener.html
AddListingListener.html
DatabaseManager.html
ListingsTableModel.html
package-frame.html
package-summary.html
package-tree.html
package-use.html
PasswordDialog.html
PhoneBookFrame.GetListener.html
PhoneBookFrame.html
PhoneDocumentListener.html
PhoneFocusListener.html
SimplePhoneBook.html
index-files
resources
allclasses-frame.html
allclasses-noframe.html
constant-values.html
deprecated-list.html
help-doc.html
index.html
overview-tree.html
C:\Documents and Settings\Steven D. Miller\workspace\java4_Lesson13\doc\greenDB
Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD
greenDB
Class DatabaseManager
java.lang.Object
greenDB.DatabaseManager
public class DatabaseManager
extends java.lang.Object
Field Summary
private java.sql.Connection
conn
connection to a database

JavaDoc

```
/**
 * Represents a student enrolled in the school.
 * A student can be enrolled in many courses.
 */
public class Student {

  /**
   * The first and last name of this student.
   */
  private String name;

  /**
   * Creates a new Student with the given name.
   * The name should include both first and
   * last name.
   */
  public Student(String name) {
    this.name = name;
  }

}
```

JavaDoc

```
/**
 * Represents a student enrolled in the school.
 * A student can be enrolled in many courses.
 */
public class Student {

  /**
   * The first and last name of this student.
   */
  private String name;

  /**
   * Creates a new Student with the given name.
   * The name should include both first and
   * last name.
   */
  public Student(String name) {
    this.name = name;
  }

}
```

JavaDoc

```
/**
 * Gets the first and last name of this Student.
 * @return this Student's name.
 */
public String getName() {
  return name;
}

/**
 * Changes the name of this Student.
 * This may involve a lengthy legal process.
 * @param newName This Student's new name.
 *                Should include both first
 *                and last name.
 */
public void setName(String newName) {
  name = newName;
}
```

JavaDoc

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
        try {
            return getImage(new URL(url, name));
        } catch (MalformedURLException e) {
            return null;
        }
}
```