JAVA 2

Sintaxe e Estrutura

ESTE MÓDULO APRESENTA A SINTAXE E ESTRUTURA DA LINGUAGEM JAVA, os operadores, expressões de controle de fluxo e a criação, declaração e utilização de vetores em Java. Destaca também as diferenças entre Java e outras linguagens.

Tópicos abordados neste módulo

- Estrutura de um programa em Java
- Sintaxe: comentários, instruções, identificadores, literais, tipos, palavras-chave.
- Convenções de programação
- Estruturas de dados
- Criação e declaração de tipos novos
- Operadores
- Expressões de controle de fluxo
- Vetores

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- Identificar as palavras-chave da linguagem Java
- Listar os oito tipos primitivos
- Saber criar um novo tipo de dados com uma classe
- Entender o que é uma classe, um método, uma variável membro e uma referência.
- Construir um novo objeto com new
- Distinguir variáveis de instância de variáveis locais
- Identificar e saber usar os operadores de Java
- Saber realizar conversões de tipos
- Saber usar as expressões de controle de fluxo
- Saber inicializar, criar, usar e copiar vetores.

1.1. Estrutura e sintaxe

As estruturas básicas de uma unidade de compilação Java: as declarações de classe (`class`), de pacote (`package`) e importações (`import`) devem aparecer no código respeitando uma determinada ordem. Se houver uma declaração `package`, do tipo:

```
package acme.java.awt;
```

ela deve ser a primeira estrutura do arquivo. Apenas comentários podem precedê-la. Essa declaração afirma que a classe (ou as classes) do arquivo de código fonte onde está contida pertencem ao referido pacote. Se um pacote não

for definido, a classe pertencerá a um pacote global que restringe-se ao diretório atual onde a classe irá executar (ela só será capaz de ser usada por outras classes que estão no mesmo diretório e que também não definam pacote).

Se houver declarações `import`, elas devem seguir a declaração `package`. A instrução `import` informa a localização de classes e não de pacotes. Por exemplo, as instruções

```
import java.awt.*;
import java.awt.event.*;
```

tornam acessíveis ao código todas as classes dos pacotes `java.awt` e `java.awt.event`. Importar todas as classes de `java.awt` não importa também o subpacote, por isto, é preciso importar suas classes explicitamente.

Após o `package` (opcional) e os `imports` (também opcionais), podem vir uma ou mais declarações e implementações de classe. A declaração deve começar com `class` ou `public class`. Se começar com `public class`, o nome do arquivo fonte (`.java`) deve ter o mesmo nome que a classe e não poderá existir outra classe no mesmo arquivo declarada com o modificador `public` (apenas classes iniciando com `class` e não `public class`). Existe em java um tipo especial de classe chamada de *interface*. A declaração poderá então iniciar com `interface` ou `public interface`, com as mesmas regras quando ao nome do arquivo fonte.

Além de `package`, `import` e `class` (ou `interface`) não poderá haver mais nada no primeiro nível do arquivo fonte Java a não ser comentários. Métodos, construtores, variáveis e constantes só podem aparecer dentro das classes e quaisquer procedimentos algorítmicos só podem estar presentes dentro de métodos, construtores e blocos `static` (um bloco especial para inicialização de valores estáticos).

As seções a seguir detalharão aspectos da sintaxe da linguagem Java e construções que poderão ser usadas nas estruturas mencionadas.

Comentários

Os comentários em Java já foram apresentados no capítulo anterior. Podem ser de dois tipos:

comentários de uma linha:

```
// isto é um comentário
```


comentários de bloco:

```
/*
 * Isto é um comentário
*/
```

Os comentários de bloco podem ainda ser usados para gerar documentação. Para isto, é necessário que sejam usados antes de classes, métodos, construtores ou variáveis e que iniciem o bloco com dois asteriscos:

```
/**
 * Este é um <i>comentário</i> de documentação.
*/
```

Comentários desse tipo aceitam vários comandos geradores de estruturas HTML e também HTML (como ilustrado acima) dentro do mesmo. Os comandos, o HTML e o texto dos comentários são usados para gerar documentação em hipertexto semelhante àquela disponível para a API (distribuída pela *Sun*). Para isto, é preciso usar a ferramenta `javadoc`.

Blocos e declarações

Uma declaração em Java consiste de uma única linha de código que termina em um ponto-e-vírgula. A declaração (a linha de código) pode ocupar múltiplas linhas do arquivo pois os espaços extras, tabulações e quebras de linha são ignorados. Tanto faz escrever:

```
System.out.println("Resultado: " + (3 * 14 / 19));
```

como escrever:

```
System.out.println
(            "Resultado: " +
        (3 *
                              14 /
               19));
```

O espaço em branco, porém, deve ser usado para tornar o código mais legível e não o contrário. Pode ser usado para endentar os blocos, ou, instruções compostas. Blocos são trechos de código entre chaves `{` e `}`. Variáveis declaradas pela primeira vez em um bloco são locais àquele bloco. Blocos são usados após declarações de classes, métodos, construtores, estruturas de controle de fluxo, etc.:


```
public class UmaClasse {
    public void umMetodo() {
       if(true) {
             System.out.println("É true!");
       }
    }
}
```

Use o espaço em branco para endentar as instruções dentro de blocos, deixar espaços verticais entre métodos e trechos longos de código, deixar espaços verticais em expressões aritméticas, com o objetivo de deixar o código mais legível.

Identificadores

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

Pode-se usar qualquer letra ou dígito do alfabeto Unicode, ou os símbolos "$" e "_" (sublinhado) em um identificador. Dígitos podem ser usados como parte de identificadores desde que não sejam o primeiro caractere.

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

Como você já observou, Java distingue letras maiúsculas de minúsculas. Isto também vale para palavras usadas como identificadores. Valor é diferente de valor que é diferente de VALOR.

Os seguintes identificadores são legais em Java:

```
Cão    variável    CLASS       R$13  índice      αντηροπος
```

Mas não são recomendadas pois podem confundir. Já pensou se você usa a palavra `índice` como variável e depois, por engano, usa `indice`? É um erro difícil de achar. `CLASS` não é palavra reservada, mas `class` é. Evite confusões! Usar **α**ντηροπος pode ser interessante se você fala grego mas imagine se você usa tais caracteres em um nome de classe executável. Como você vai conseguir rodar o programa se não tiver as fontes correspondentes no seu sistema?

Esses outros identificadores são ilegais:

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

Palavras-chave

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

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

Apesar de reservadas, as palavras `const`, e `goto`, e não são usadas atualmente (Java 2).

Os nomes de classes fundamentais da linguagem não são consideradas palavras reservadas, mas você deve evitar usá-las. Se você cria uma variável chamada `String` você terá erros de tipo incorreto. Se criar uma classe chamada `String` terá erros de ambigüidade a não ser que o seu programa crie um pacote específico para essa sua classe. Se puder evitar, porém, evite usar nomes de classes dos pacotes Java.

Literais

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

- `12.4` (12,4 ponto flutuante decimal)
- `0377` (377 inteiro octal)
- `0xff991a` (FF991A inteiro hexadecimal)
- `true` (literal booleana)

- `'a'` (caractere)
- `"barata"` (cadeia de caracteres)

Numéricos

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

```
12.4  0xab779c .27777e-23      0137  20  Double.NaN
```

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

Booleanos

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

Caracteres e strings

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

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

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

```
"anta"  "vampiros são morcegos"     "" (vazia)
"\u3F07\u3EFA \u3F1C"               " " (espaço)
"Uma linha\nDuas Linhas\tTabulação"
```

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

SEQÜÊNCIA	VALOR DO CARACTERE
\b	Retrocesso (backspace)
\t	Tabulação
\n	Nova Linha (new line)
\f	Alimentação de Formulário (form feed)
\r	Retorno de Carro (carriage return)
\"	Aspas
\'	Aspa
\\	Contra Barra
nnn	O caractere correspondente ao valor octal *nnn*, onde *nnn* é um valor entre 000 e 0377.
\u*nnnn*	O caractere Unicode *nnnn*, onde *nnnn* é de um a quatro dígitos hexadecimais. Seqüências Unicode são processadas antes das demais seqüências.

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

Você pode usar identificadores com acentos e cedilha para dar nomes a qualquer variável, método ou classe, pois Java os converte para Unicode antes de compilar (porém evite fazer isso no caso dos nomes das classes, já que os nomes de arquivo gerados são dependentes do sistema operacional).

Para representar aspas e contra-barras dentro de um literal String ou de caractere, é necessário usar um escape também, precedendo o caractere por uma outra contra-barra, por exemplo, para imprimir a linha:

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

é preciso usar:

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


Tipos primitivos

Java acrescenta `byte` e `boolean` ao conjunto de tipos de dados primitivos das linguagens C e C++. A principal diferença em relação a C ou C++ é que os *tamanhos* em bytes dos tipos em Java são definidos, e não dependem da plataforma, o que não acontece naquelas duas outras linguagens. Além disso, em C, quando uma variável é usada sem ser inicializada, geralmente contém lixo. Java define valores *default* para todos os tipos, que são usados caso uma variável não tenha sido inicializada na criação do objeto a qual pertence.

Variável alguma em Java tem permissão para ter um valor não definido. Variáveis locais têm que ser inicializadas antes de serem usadas ou um erro de compilação irá ocorrer. A primeira inicialização de uma variável local também não pode ser feita dentro de um bloco `if` a não ser que ela própria tenha sido declarada dentro desse bloco (é local ao `if`). Se isto ocorrer, o compilador acusará o erro mais adiante quando a variável for usada e dirá que "a variável pode não ter sido inicializada). Variáveis definidas dentro do bloco de uma declaração de classe (membros – de instância e de classe) são inicializadas com valores *default*.

A tabela abaixo relaciona os tipos primitivos em Java, seus tamanhos, faixa de valores dos seus literais e valores *default*.

TIPO DE DADOS	TAMANHO	VALORES	DEFAULT
`boolean`	8 bits	`true` ou `false`	`false`
`byte`	8 bits	-128 a 127	0
`short`	16 bits	-32768 a 32767	0
`char`	16 bits	\u0000 a \uffff	\u0000
`int`	32 bits	-2147483648 a 2147483648	0
`long`	64 bits	-9223372036854775808 a 9223372036854775807	0L
`float`	32 bits	1,40239846e-45 a 3,40282347e38	0.0f
`double`	64 bits	4,94065645841264544e-324 a 1.79769313486231570e308	0.0d

Qualquer identificador válido em Java pode seru usado para dar nome a uma variável. Todas as variáveis em Java devem ser *declaradas* antes de serem usadas. Se uma variável for declarada como sendo de um determinado tipo receber um valor de outro tipo, ela poderá ter que ser convertida explicitamente

através de uma conversão forçada (*cast*), também chamada de coerção. Variáveis de tipos menores podem, em geral, terem seus valores atribuídos a tipos maiores sem coerção. O contrário exige que o tipo (resultante) seja informado entre parênteses. Se o valor armazenado for maior que o que o tipo final pode comportar, haverá perda de informação.

```
float f;
int i;

f = 3.14;
i = (int)f; // i recebe f, que é convertido para int
            // (números depois da vírgula são truncados)
```

A conversão ou coerção só é possível entre tipos numéricos e caracteres. Booleanos são inconversíveis. Uma discussão maior sobre conversão e coerção será apresentada mais adiante.

Tipos boolean

Valores booleanos não são inteiros e nem podem ser tratados como tal. Também não é possível converter valores booleanos em outros tipos através de atribuição. Para traduzir valores booleanos em inteiros, pode-se usar um bloco `if` ou o operador ternário:

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

Tipos char

Tipos `char` contém um caractere Unicode de dois bytes. O primeiro `byte` corresponde ao ASCII ou ISO-Latin1.

A conversão de `char` em `int` ou `long` obtém o valor Unicode do caractere. Os tipos `char` não tem sinal. Se um `char` (16 bit) for convertido para um tipo `byte` (8 bit) ou `short` (16 bit), pode resultar em um valor negativo.

Um `char` sempre pode ser manipulado como um `int`. Se for necessário representar o caractere, porém, é preciso convertê-lo (através de um cast) para `char`:

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


Tipos inteiros

Todos os tipos inteiros tem sinal (não existe a palavra-chave `unsigned` como em C). Inteiros podem ser do tipo `byte` (8 bit), `short` (16 bit), `int` (32 bit) ou `long` (64 bit). A coerção de tipos de tamanhos maiores em tipos menores pode resultar em perda de dados caso os valores a serem convertidos sejam maiores que os valores máximos armazenados por cada tipo.

A representação *default* para inteiros é `int`. Se uma atribuição para uma variável do tipo `long` recebe um `int`, este é automaticamente convertido (e ampliado) para long. Se o literal for um número maior que o valor máximo de um `int`, porém, o número será `long` e provocará erro ao ser atribuído a uma variável do tipo `int` a não ser que seja através de uma coerção (que provocará perda de informação). Pode-se distinguir literais do tipo `long` das constantes `int` utilizando o caractere "L" como sufixo (ex: `127L`). Evite, porém, usar o "`L`" minúsculo (é permitido, mas como ele se parece muito com o algarismo "`1`", poderá ser fonte de confusão).

A divisão de um número inteiro por zero sempre causa uma exceção (erro em tempo de execução) do tipo `ArithmeticException`.

Tipos de ponto-flutuante

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

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

Nenhuma operação de ponto flutuante provoca exceções. Divisões por zero provocam números infinitos ou indeterminações identificadas pelas constantes `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY` e `Float.NaN`, que representam respectivamente, infinito positivo, infinito negativo e "não é número".

Convenções

Java possui várias convenções de codificação que visam facilitar a programação e principalmente a análise do código. Essas convenções são adotadas em toda a documentação. O seu uso é opcional mas recomendado.

Nomes de classes e interfaces

Os nomes de classes são substantivos (são coisas) com caixa mista (cada palavra começa com maiúsculas). Por exemplo:

```
class LivroDeReceitas
class Circulo
```

Nomes de métodos

Os nomes de métodos devem ser verbos (representam ação) com caixa mista (como as classes), mas, sempre com a primeira letra em caixa baixa:

```
public void imprimirRelatorio()
```

Se você encontrar na API Java métodos com a primeira letra maiúscula observe bem e concluirá que é um construtor. Construtores têm sempre o mesmo nome que a classe que o contém (portanto, primeira letra maiúscula) e não possuem tipo de retorno (`void`, `int`, `String`).

Nomes de variáveis

Assim como os métodos, variáveis devem aparecer em caixa mista com a primeira letra minúscula.

```
int moscaVerde;
```

Constantes

Se você encontrar variáveis com todas as letras maiúsculas na API Java, são constantes escalares. As constantes têm todas as letras maiúsculas e se forem formadas por múltiplas palavras, essas palavras devem ser separadas por sublinhado:

```
public final int UMA_CONSTANTE = 15;
```

Constantes não-escalares (objetos) têm caixa mista como as variáveis comuns (pois as suas referências não são constantes). Por exemplo: `red` e `blue` são constantes da classe `Color`.

Nomes de pacotes

Pacotes são representados com todas as letras minúsculas (mesmo que contenham mais de uma palavra)

1.2. Classes e objetos

Estruturas

Suponha que, em determinado programa em Java, você precise especificar um círculo. O círculo pode ser totalmente especificado com um raio e as coordenadas do seu centro. Usando os tipos primitivos de Java, podemos especificar um círculo, dentro de um método, da forma:

```
double x0;
double y0;
double raio;
```

E então, para usar este círculo, bastaria definir cada uma das variáveis:

```
x0 = 1.2;
y0 = 0.8;
raio = 1.5;
```

Assim, poderemos desenhá-lo em uma determinada posição da tela, usando a equação $(x - x_0)^2 + (y - y_0)^2 = raio^2$ e variando x e y.

Suponha agora que o nosso programa exige que lidemos com vários círculos ao mesmo tempo. Teremos que ter três variáveis diferentes para cada círculo e repetir o mesmo processo várias vezes. Só para representar desenhar 100 círculos, teríamos que fazer:

```
double xCirc1, yCirc1, raioCirc1;
double xCirc2, yCirc2, raioCirc2;
(...)
double xCirc100, yCirc100, raioCirc100;
```

e teríamos que especificar cada tipo um por um. Podemos, é claro, usar vetores. A desvantagem é que o vetor provavelmente seria usado para agrupar partes de cada círculo como, por exemplo, todas as coordenadas x, todos os raios, etc. Não poderíamos errar na ordem de um vetor que tinha que sempre estar alinhado com o outro. E se o objeto fosse mais complicado (tivesse mais que três propriedades e de tipos diferentes)? E se estes círculos fossem também

propriedades de um objeto mais complexo? A solução, talvez, seja criar um novo tipo de dados!

Tipos de dados abstratos

Várias linguagens, mesmo as que não são orientadas a objetos, permitem que o programador crie *tipos de dados abstratos*, ou *estruturas*. Em Java, podemos fazer isto com uma *classe*, que é muito mais que uma mera estrutura, como veremos no capítulo seguinte. A estrutura define somente um estado, uma coleção de propriedades. Não define *como* os seus dados podem ser transformados (métodos). Uma classe Java pode conter não só os tipos básicos que a definem, mas também os métodos que caracterizam o comportamento dos objetos criados a partir dela.

Criação de um novo tipo

Para produzir um novo tipo de dados *Círculo* em Java, com as propriedades x_0, y_0 e *raio*, poderíamos definir:

```
public class Circulo extends Object {
       public double    x0,
                        y0,
                        raio;
}
```

Isto é lido da seguinte forma: "Um Círculo *é um* objeto que *tem uma* coordenada x, uma coordenada y e um raio". Lembre-se que a parte `extends Object` é opcional e geralmente é omitida por ser implícita na declaração de classes Java.

Agora, podemos tratar a classe recém-criada como um novo *tipo de dados* e, dentro de um método ou construtor de outra classe (possivelmente uma classe executável) declarar variáveis como *referências* para estes círculos:

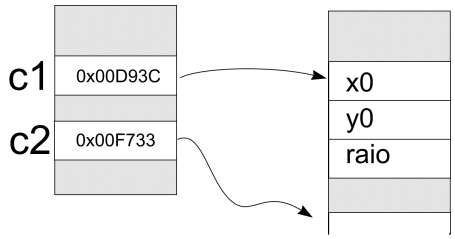
```
Circulo c1, c2, c3;
```

Ao fazer isto, não criamos novos objetos, apenas alocamos memória para as suas *referências* ou *ponteiros*. Estas referências contém o endereço de memória dos objetos. O endereço é definido em tempo de execução e, portanto, não se pode realizar aritmética com ele como em C ou C++.

Criação de um objeto

Para efetivamente *criar* um objeto, é necessário alocar memória para ele usando o operador "new", da forma:

```
c1 = new Circulo();
c2 = new Circulo();
```



Observe que o gerenciamento de memória para os *tipos de objetos* ocorre de forma diferente dos *tipos primitivos*. Na declaração de um `int`, a memória para o seu armazenamento é imediatamente alocada. O sistema sabe que um int ocupa 32 bits. No caso de objetos, a alocação de memória só é possível através do operador `new` que chama um procedimento (construtor) que irá determinar quando espaço será necessário para o objeto e alocá-lo.

Depois de criado o objeto, Os valores iniciais de seus campos de dados são inicializados a zero (ou valores equivalentes, de acordo com o tipo). Somente depois de criado o objeto, podem os campos de cada circulo ser acessados usando o operador "." (ponto):

```
c1.x0 = 1.2;
c1.y0 = 0.8;
double x = c1.x0 + 2;
```

Qualquer classe, portanto, representa um *tipo* de dados e qualquer variável de classe, de instância ou local pode ser declarada como sendo do "tipo" de uma determinada classe da mesma maneira que se declara tipos primitivos. Por exemplo:

```
long preço;  // preço é variável do tipo primitivo long
String nome; // nome é variável do tipo objeto String
Círculo c;   // c é variável do tipo objeto Círculo
```

Referências

Nenhuma declaração *cria* objetos mas apenas *declara* uma variável como tendo o tipo definido em sua classe (`String` ou `Círculo` no exemplo acima), preparando a variáveis para que seja uma *referência* a um objeto desse tipo.

A declaração

```
long preço;
```


estabelece que a variável `preço` está preparada para receber um valor numérico inteiro primitivo do tipo `long` (aloca 64 bits de memória para acomodar o `long`). Após a declaração, é comum ter uma atribuição, *definindo* o valor da variável, por exemplo:

```
preço = 35700;
```

É simples. A criação de um objeto é mais complexa. O sistema não sabe quanto espaço será necessário para acomodar um `Círculo` e então só poderá alocar o espaço necessário para acomodar a referência (32 ou 64 bits).
No nosso exemplo, a alocação da memória para o objeto `Circulo` vai exigir pelo menos 3 vezes 64 = 192 bits, referentes aos campos x, y e raio, sem contar o espaço necessário para os métodos herdados. Também é preciso chamar um construtor que irá dizer como se constrói o objeto. A variável declarada recebe, por atribuição, uma referência (ponteiro) para este objeto:

```
c = new Circulo();
```

`Circulo()` é o construtor e `new`, o operador de alocação de memória usado para criar novos objetos. Depois do `new`, a memória foi alocada, e c é agora uma referência para a posição de memória onde está o objeto que já pode ser usado. A tentativa de usar um objeto antes de sua criação (com `new`) provoca uma exceção em tempo de execução (`NullPointerException`).

Membros de um objeto

As váriáveis e métodos definidos no interior de uma classe são considerados membros de um objeto desde que não possuam o modificador static na sua declaração. Por membros de um objeto queremos dizer que variáveis e métodos fazem parte do objeto criado. Uma classe (tipo) pode ser usada para declarar várias variáveis. Cada uma, porém, ao ser inicializada com um construtor, aponta para um objeto diferente, que possui as variáveis e métodos definidos na classe. Os métodos e variáveis só podem ser usados pelos objetos e não pelas classes. As classes são meras plantas ou moldes usados para moldar ou construir objetos reais que têm propriedades e que podem realizar ações.

Por exemplo, a classe:

```
class UmObjeto {
   private int ox;
   private static int cx;
```



```
    public static void clmet() { ... }
    public void obmet() { ... }
}
```

tem duas variáveis e dois métodos. Os objetos criados com essa classe terão apenas uma variável e um método, que são seus membros. Os outros dois são membros da classe, pois foram declarados com `static`. Só há uma classe, portanto, só há uma cópia da variável `cx`. A variável `ox` e o método `obmet()` só podem ser usados através de objetos. Já o método `clmet()` e a variável cx podem ser usadas de qualquer lugar dentro da classe. Um método da própria classe `UmObjeto` pode criar instâncias (objetos) a partir de si próprio:

```
public static void clmet() {
   UmObjeto obj = new UmObjeto();
   obj.ox = 13;
   obj.obmet();
   cx = 15;
}
```

Um método estático não pode chamar, porém, variáveis de instância pertencentes ao objeto a não ser que crie o objeto como acima:

```
public static void clmet() {
   ox = 13;  // dá ERRO de compilação
   obmet();  // dá ERRO de compilação
   cx = 15; // OK. É static.
}
```

Portanto, `ox` e `obmet()` são *membros* dos objetos criados através da classe `UmObjeto`.

1.3. Expressões e controle de fluxo

Esta seção apresentará os operadores usados na linguagem Java e as estruturas básicas de dados.

Operadores

Operadores permitem a realização de tarefas como adição, subtração, multiplicação, atribuição, etc. Podem ser divididos em três categorias: operadores booleanos, operadores de atribuição e operadores numéricos. Os operadores de Java são quase os mesmos de C/C++, com algumas ausências.

A tabela abaixo lista os operadores usados na linguagem Java:

OPERADOR	FUNÇÃO	OPERADOR	FUNÇÃO
+	Adição	~	complemento
-	Subtração	<<	deslocamento à esquerda
*	Multiplicação	>>	deslocamento à direita
/	Divisão	>>>	desloc. a direita com zeros
%	Resto	=	atribuição
++	Incremento	+=	atribuição com adição
--	Decremento	-=	atribuição com subtração
>	Maior que	*=	atribuição com multiplicação
>=	Maior ou igual	/=	atribuição com divisão
<	Menor que	%=	atribuição com resto
<=	Menor ou igual	&=	atribuição com AND
==	igual	|=	atribuição com OR
!=	não igual	^=	atribuição com XOR
!	NÃO lógico	<<=	atribuição com desloc. esquerdo
&&	E lógico	>>=	atribuição com desloc. direito
||	OU lógico	>>>=	atrib. C/ desloc. a dir. c/ zeros
&	AND	?:	Operador ternário
^	XOR	(*tipo*)	Conversão de tipos
|	OR	instanceof	Comparação de tipos

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

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

ASSOC	TIPO DE OPERADOR	OPERADOR
D a E	separadores	`[] . ; , () (`
E a D	operadores unários	`new (cast) expr++ expr-- ++expr --expr +expr -expr ~ !`
E a D	multiplicativo	`* / %`
E a D	aditivo	`+ -`
E a D	deslocamento	`<< >> >>>`
E a D	relacional	`< > >= <= instanceof`
E a D	igualdade	`== !=`
E a D	AND	`&`
E a D	XOR	`^`
E a D	OR	`|`
E a D	E lógico	`&&`
E a D	OU lógico	`||`
D a E	condicional	`?:`
D a E	atribuição	`= += -= *= /= %= >>= <<= >>>= &= ^= !=`

Observe que a comparação é realizada com um sinal de "=" duplo. O uso de um único sinal "=" caracteriza uma atribuição.

Também são bastante usados os operadores unários "++" e "−" para incrementar (somar 1) ou decrementar (subtrair 1), respectivamente, uma variável. Estes operadores podem alterar o valor da variável antes ou depois que ela for usada.

Java sobrecarrega o operador "+" para realizar a concatenação de `Strings`, além da adição de valores numéricos.

As expressões a seguir são válidas em Java:

```
 1 + 2
(1 + 2) * i
(a > 0) && (a < 1)
 a == 2
```

As duas primeiras expressões são do tipo de `i` (se `i` for inteiro, o resultado é inteiro). Algumas expressões, porém, podem resultar em conversões de tipos, como vimos anteriormente. A terceira e quarta resultam em `true` ou `false` (resultado tipo `boolean`).

Conversão e coerção

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

Para fazer essas conversões "ilegais", pode-se usar o operador de coerção (cast). Com ele, o risco fica com o programador que diz ao compilador que sabe que pode perder dados. Ele diz isto colocando o tipo a ser convertido entre parênteses antes da expressão.

Há quatro tipos de conversões entre tipos primitivos: por atribuição, por passagem de parâmetro em método, por promoção aritmética e por casting. A conversão por *atribuição* é legal quando o tipo do lado esquerdo é maior em número de bytes ou e precisão que o tipo do lado direito:

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



```
i = d; // ilegal!!!
```

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

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

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

```
int x = 5;
long ab = 10;
double = x + ab + 3.25D;
```

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

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

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

```
int i = (byte) (b1 + b2); // b1 e b2 são do tipo byte.
```

Controle de fluxo

As expressões de controle de fluxo são basicamente as mesmas do C ou C++. Uma diferença fundamental é que os resultados das expressões usadas como teste devem ser obrigatoriamente booleanas (resultar em `true` ou `false`).

Seleção if ... else

Sintaxe básica:

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

O primeiro `if` é obrigatório. Os outros blocos são opcionais. Se a expressão entre parênteses resultar em `true`, o código entre `{` e `}` será executado, caso contrário o bloco `else` é executado, se existir. *Exemplo:*

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

Seleção while, do...while

Sintaxe básica:

```
while (expressão) {...}    ou
do {...} while (expressão);
```

Se o resultado da expressão for `true`, o bloco de `while` será executado. Na seleção `do... while`, o bloco é executado pelo menos uma vez antes da expressão ser testada.

Exemplo:

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

Iteração for

Sintaxe básica:

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

`inicial`, `teste` e `incremento` são todas expressões que controlam o loop. Qualquer uma delas ou todas são opcionais.

Exemplo:

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

Seleção switch

Seleção tipo *switch-case*. É idêntica à expressão `switch` usada em C ou C++. *Sintaxe básica:*

```
switch (expressão) {
```



```
    case constante_1 : expressões; break;
    case constante_2 : expressões; break;
    ...
    case constante_n : expressões; break;
    default: expressões; break;
}
```

Na expressão acima, o `break` é essencial para sair de uma cláusula `case`. A única exceção é se houver um `return` ou ocorrer uma exceção (`Exception`) no código. Se não houver `break` no fim de um dos `cases`, o controle passará ao bloco seguinte e assim por diante até que o `switch` acabe ou um `break` seja encontrado.

Há ainda outras formas de interferir no fluxo de um loop (`for` ou `while`), com `break`, `continue` e rótulos (`labels`): `break` pode ser usada para forçar a saída do loop e `continue` pode ser usado para sair de um loop de vários níveis e recomeçar em um nível mais externo, identificado por um rótulo (um identificador, seguido de ":"). É semelhante ao uso em JavaScript, mas em Java pode-se definir rótulos.

1.4. Vetores

Não existe classe especial para vetores. No entanto, eles também não são tipos primitivos. Vetores (*arrays*) têm, em Java, *status* de objeto, mas não têm uma classe correspondente. Todo vetor possui uma variável pública chamada `length`, que informa o tamanho do mesmo. É o único meio de obter o tamanho de um vetor em Java. No trecho de código abaixo, a variável `x` armazena o valor 4, que corresponde ao comprimento do vetor.

```
int vetor[] = {1, 1, 2, 2};
int x = vetor.length;
```

Além da maneira acima, existem outras formas de declarar e inicializar vetores. A inicialização utiliza a palavra-chave "new" e define a quantidade de elementos do vetor (que devem ter o mesmo tipo), como mostrado nos exemplos abaixo. Também pode-se definir vetores de vetores, que na prática são vetores *multidimensionais*.

Exemplos:

Declaração de um vetor de `Strings` unidimensional;

```
String[] args;
```



```
String args[]; // sintaxe opcional (semelhante a C)
```

Declaração e inicialização de um vetor de `Strings` unidimensional;

```
String[] args = new String[10]; // sintaxe mais clara!
String args[] = new String[10];
```

Declaração de três vetores de inteiros: um tridimensional (`espaço`), um bidimensional (`matriz`) e um unidimensional (`vetor`). Exemplos de sintaxes variadas:

```
int espaço[][][], matriz[][], vetor[];
int[] espaço[][], matriz[], vetor;
int[][] espaço[], matriz; int[] vetor;
int[][][] espaço; int[][] matriz; int[] vetor; // sintaxe mais clara!!!
```

Inicialização de um vetor bidimensional e de um tridimensional:

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

Inicialização explícita (só é permitida durante a declaração):

```
int[][] matriz = {{0, 1}, {0, 1, 2}};
```

