VARIÁVEIS E TIPOS DE DADOS

FUNDAMENTOS DE PROGRAMAÇÃO

As palavras-chave java

Sessenta palavras-chave estão atualmente definidas na linguagem Java. Estas palavras-chave, combinadas com a sintaxe dos operadores e separadores, formam a definição da linguagem Java. Em geral, as palavras-chave não podem ser usadas como nomes para uma variável, classe ou método. As exceções a essa regra são as palavras-chave context-sensitive (sensível ao contexto) adicionadas pelo JDK 9 para suportar módulos.
Além disso, a partir do JDK 9, um sublinhado por si só é considerado uma palavra-chave, a fim de evitar seu uso como o nome de algo em seu programa.

As palavras-chave const e goto são reservadas, mas não usadas. Nos primórdios do Java, várias outras palavras-chave foram reservadas para possível uso futuro. No entanto, a especificação atual para Java define apenas as palavras-chave mostradas na tabela mostrada no slide a seguir.

Além das palavras-chave, Java reserva outros quatro nomes. Três fazem parte do Java desde o início: true, false e null. Estes são valores definidos pelo Java. Você não pode usar essas palavras para os nomes de variáveis, classes e assim por diante. A partir do JDK 10, a palavra var foi adicionada como um nome de tipo reservado sensível ao contexto.

As palavras-chave java

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	exports	extends
final	finally	float	for	goto	if
implements	import	instanceof	int	interface	long
module	native	new	open	opens	package
private	protected	provides	public	requires	return
short	static	strictfp	super	switch	synchronized
this	throw	throws	to	transient	transitive
try	uses	void	volatile	while	with
_					

Identificadores em java

Em Java, um identificador é, essencialmente, um nome dado a um método, uma variável ou qualquer outro item definido pelo utilizador. Os identificadores podem ter de um a vários caracteres. Os nomes das variáveis podem começar com qualquer letra do alfabeto, um sublinhado ou um cifrão. O próximo pode ser uma letra, um dígito, um cifrão ou um sublinhado. O sublinhado pode ser usado para melhorar a legibilidade de um nome variável, como em line_count. Maiúsculas e minúsculas são diferentes; ou seja, para Java, myvar e MyVar são nomes separados. Eis alguns exemplos de identificadores aceitáveis:

Test	x	y2	MaxLoad
$up	_top	my_var	sample23

Lembre-se, não é possível iniciar um identificador com um dígito. Assim, 12x é inválido, por exemplo.

Em geral, não é possível usar as palavras-chave Java como nomes de identificadores. Além disso, você não deve usar o nome de nenhum método padrão, como println, como um identificador. Além dessas duas restrições, as boas práticas de programação determinam que você use nomes de identificador que reflitam o significado ou o uso dos itens que estão sendo nomeados.

As bibliotecas de classes java

Os programas de exemplo mostrados neste capítulo fazem uso de dois dos métodos incorporados do Java: println( ) e print( ).

Esses métodos são acessados através de System.out. System é uma classe predefinida por Java que é automaticamente incluída em seus programas.

Na visão maior, o ambiente Java depende de várias bibliotecas de classe incorporadas que contêm muitos métodos internos que fornecem suporte para coisas como E/S, manipulação de cadeia de caracteres, rede e gráficos.

As classes padrão também fornecem suporte para uma interface gráfica do usuário (GUI).

Assim, Java como uma totalidade é uma combinação da própria linguagem Java, mais suas classes padrão. Como verá, as bibliotecas de classe fornecem grande parte do que vem com Java.

De fato, parte de se tornar um programador Java é aprender a usar as classes Java padrão. Ao longo deste livro, vários elementos das classes e métodos padrão da biblioteca são descritos. No entanto, a biblioteca Java é algo que você também vai querer explorar mais por conta própria.

Por que os tipos de dados são importantes

Os tipos de dados são especialmente importantes em Java porque é uma linguagem fortemente tipada. Isso significa que todas as operações são verificadas pelo compilador quanto à compatibilidade de tipo.

As operações ilegais não serão compiladas. Assim, a verificação de tipo forte ajuda a evitar erros e aumenta a confiabilidade.

Para habilitar a verificação de tipo forte, todas as variáveis, expressões e valores têm um tipo. Não existe o conceito de uma variável "sem tipo", por exemplo.

Além disso, o tipo de um valor determina quais operações são permitidas nele. Uma operação permitida em um tipo pode não ser permitida em outro.

Tipos primitivos de java

Java contém duas categorias gerais de tipos de dados incorporados: orientado a objetos e não orientado a objetos. Os tipos orientados a objetos do Java são definidos por classes, e uma discussão de classes é adiada para mais tarde.

No entanto, no núcleo do Java estão oito tipos primitivos (também chamados elementares ou simples) de dados, que são mostrados no slide a seguir.

O termo primitivo é usado aqui para indicar que esses tipos não são objetos em um sentido orientado a objetos, mas sim, valores binários normais.

Esses tipos primitivos não são objetos por questões de eficiência. Todos os outros tipos de dados do Java são construídos a partir desses tipos primitivos.

Tipos primitivos de java

Type	Meaning
boolean	Represents true/false values
byte	8-bit integer
char	Character
double	Double-precision floating point
float	Single-precision floating point
int	Integer
long	Long integer
short	Short integer

Números inteiros

Java define quatro tipos inteiros: byte, short, int e long, que são mostrados aqui:

Type	Width in Bits	Range
byte	8	–128 to 127
short	16	–32,768 to 32,767
int	32	–2,147,483,648 to 2,147,483,647
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Como mostra a tabela, todos os tipos inteiros são assinados valores positivos e negativos. Java não suporta inteiros não assinados (positiveonly). Muitas outras linguagens de computador suportam inteiros assinados e não assinados. No entanto, os designers do Java sentiram que inteiros não assinados eram desnecessários.

Observação

Tecnicamente, o sistema de tempo de execução Java pode usar qualquer tamanho que queira para armazenar um tipo primitivo. No entanto, em todos os casos, os tipos devem agir conforme especificado.

O tipo inteiro mais comumente usado é int. Variáveis do tipo int são frequentemente empregadas para controlar loops, indexar matrizes e executar matemática inteira de uso geral.

Quando precisar de um número inteiro que tenha um intervalo maior que int, use long. Por exemplo, aqui está um programa que calcula o número de polegadas cúbicas contidas em um cubo que é uma milha por uma milha, por uma milha:

```
/*
   Compute the number of cubic inches
   in 1 cubic mile.
*/
class Inches {
  public static void main(String args[]) {
    long ci;
    long im;

    im = 5280 * 12;

    ci = im * im * im;

    System.out.println("There are " + ci +
                       " cubic inches in cubic mile.");

  }
}
```

Observação

O menor tipo de inteiro é byte.

As variáveis do tipo byte são especialmente úteis ao trabalhar com dados binários brutos que podem não ser diretamente compatíveis com outros tipos incorporados do Java.

O tipo curto cria um inteiro curto. Variáveis do tipo short são apropriadas quando você não precisa do intervalo maior oferecido pelo int.

```
/*
   Compute the number of cubic inches
   in 1 cubic mile.
*/
class Inches {
  public static void main(String args[]) {
    long ci;
    long im;

    im = 5280 * 12;

    ci = im * im * im;

    System.out.println("There are " + ci +
                       " cubic inches in cubic mile.");

  }
}
```

Tipos de ponto flutuante

Os tipos de ponto flutuante podem representar números que têm componentes fracionários. Existem dois tipos de ponto flutuante, float e double, que representam números de precisão simples e dupla, respectivamente. O tipo float tem 32 bits de largura e o tipo double tem 64 bits de largura.

Das duas, double é a mais comumente usada, e muitas das funções matemáticas na biblioteca de classes do Java usam valores duplos. Por exemplo, o método sqrt( ) (que é definido pela classe Math padrão) retorna um valor duplo que é a raiz quadrada de seu argumento duplo.

Aqui, sqrt( ) é usado para calcular o comprimento da hipotenusa, dados os comprimentos dos dois lados opostos:

```
/*
   Use the Pythagorean theorem to
   find the length of the hypotenuse
   given the lengths of the two opposing
   sides.
*/
class Hypot {
  public static void main(String args[]) {
    double x, y, z;

    x = 3;
    y = 4;
    z = Math.sqrt(x*x + y*y);

    System.out.println("Hypotenuse is " +z);
  }
}
```

Notice how **sqrt( )** is called. It is preceded by the name of the class of which it is a member.

Um outro ponto sobre o exemplo anterior: Como mencionado, sqrt( ) é um membro da classe padrão de Matemática. Observe como sqrt( ) é chamado; é precedido pelo nome Math. Isso é semelhante à maneira como System.out precede println( ). Embora nem todos os métodos padrão sejam chamados especificando seu nome de classe primeiro, vários são.

Characters

Em Java, os caracteres não são quantidades de 8 bits como em muitas outras linguagens de computador. Em vez disso, Java usa Unicode.

Unicode define um conjunto de caracteres que pode representar todos os caracteres encontrados em todos os idiomas humanos. Em Java, char é um tipo de 16 bits não assinado com um intervalo de 0 a 65.535. O conjunto de caracteres ASCII padrão de 8 bits é um subconjunto de Unicode e varia de 0 a 127. Assim, os caracteres ASCII ainda são caracteres Java válidos.

A uma variável de caractere pode ser atribuído um valor colocando o caractere entre aspas simples.
Por exemplo, isso atribui à variável ch a letra X:

```
char ch;
ch = 'X';
```

Characters

```
// Character variables can be handled like integers.
class CharArithDemo {
  public static void main(String args[]) {
    char ch;

    ch = 'X';
    System.out.println("ch contains " + ch);

    ch++; // increment ch <-- A char can be incremented.
    System.out.println("ch is now " + ch);

    ch = 90; // give ch the value Z <-- A char can be assigned an integer value.
    System.out.println("ch is now " + ch);
  }
}
```

No programa, ch recebe primeiro o valor X. Em seguida, ch é incrementado. Isso resulta em ch contendo Y, o próximo caractere na sequência ASCII (e Unicode).

Em seguida, ch é atribuído o valor 90, que é o valor ASCII (e Unicode) que corresponde à letra Z. Como o conjunto de caracteres ASCII ocupa os primeiros 127 valores no conjunto de caracteres Unicode, todos os "truques antigos" que você pode ter usado com caracteres em outras linguagens também funcionarão em Java.

O tipo booleano

O tipo booleano representa valores verdadeiros/falsos. Java define os valores true e false usando as palavras reservadas true e false. Assim, uma variável ou expressão do tipo booleano será um desses dois valores.

Aqui está um programa que demonstra o tipo booleano:

```
// Demonstrate boolean values.
class BoolDemo {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

Literais

Os literais Java podem ser de qualquer um dos tipos de dados primitivos. A forma como cada literal é representado depende do seu tipo. Como explicado anteriormente, as constantes de caracteres são colocadas entre aspas simples. Por exemplo, 'a' e '%' são constantes de caracteres. Literais inteiros são especificados como números sem componentes fracionários. Por exemplo, 10 e –100 são literais inteiros. Os literais de ponto flutuante requerem o uso do ponto decimal seguido pelo componente fracionário do número. Por exemplo, 11.123 é um literal de ponto flutuante. Java também permite que você use notação científica para números de ponto flutuante.

Por padrão, literais inteiros são do tipo int. Se quiser especificar um literal longo, acrescente um l ou um L. Por exemplo, 12 é um int, mas 12L é um longo.

Por padrão, literais de ponto flutuante são do tipo double. Para especificar um literal flutuante, acrescente um F ou f à constante. Por exemplo, 10.19F é do tipo float.

Literais hexadecimais, octais e binários

Como você deve saber, na programação às vezes é mais fácil usar um sistema de números baseado em 8 ou 16 em vez de 10. O sistema numérico baseado em 8 é chamado octal, e usa os dígitos de 0 a 7. No octal o número 10 é o mesmo que 8 em decimal.

O sistema de números de base 16 é chamado hexadecimal e usa os dígitos de 0 a 9 mais as letras de A a F, que significam 10, 11, 12, 13, 14 e 15. Por exemplo, o número hexadecimal 10 é 16 em decimal. Devido à frequência com que esses dois sistemas numéricos são usados, o Java permite especificar literais inteiros em hexadecimal ou octal em vez de decimal. Um literal hexadecimal deve começar com 0x ou 0X (um zero seguido de um x ou X). Um literal octal começa com um zero. Eis alguns exemplos:

```
hex = 0XFF // 255 em decimal
Out = 011 // 9 pol decimal
```

Sequências de fuga de personagens/ Character

Colocar constantes de caracteres entre aspas simples funciona para a maioria dos caracteres de impressão, mas alguns caracteres, como o retorno de carro, representam um problema especial quando um editor de texto é usado.

Além disso, alguns outros caracteres, como aspas simples e duplas, têm um significado especial em Java, portanto, você não pode usá-los diretamente. Por esses motivos, o Java fornece sequências de escape especiais, às vezes chamadas de constantes de caracteres de barra invertida, mostradas na Tabela. Essas sequências são usadas no lugar dos caracteres que representam.

Escape Sequence	Description
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line
\f	Form feed
\t	Horizontal tab
\b	Backspace
ddd	Octal constant (where *ddd* is an octal constant)
\u*xxxx*	Hexadecimal constant (where *xxxx* is a hexadecimal constant)

Literais de string

Java suporta um outro tipo de literal: a string. Uma string é um conjunto de caracteres entre aspas duplas. Por exemplo, é uma string.

Você viu exemplos de strings em muitas das instruções println( ) nos programas de amostra anteriores.

Além dos caracteres normais, uma string literal também pode conter uma ou mais das sequências de escape que acabamos de descrever.

Por exemplo, considere o seguinte programa. ele usa as seqüências de escape \n e \t.

```
// Demonstrate escape sequences in strings.
class StrDemo {
  public static void main(String args[]) {
    System.out.println("First line\nSecond line");
    System.out.println("A\tB\tC");
    System.out.println("D\tE\tF") ;
  }
}
```

Use \n to generate a new line.

Use tabs to align output.

Um olhar mais atento sobre as variáveis

Aqui, vamos analisá-los mais de perto. Como você aprendeu anteriormente, as variáveis são declaradas usando esta forma de instrução, type varname;

onde type é o tipo de dados da variável e varname é seu nome. Você pode declarar uma variável de qualquer tipo válido, incluindo os tipos simples descritos acima, e cada variável terá um tipo.

Assim, as capacidades de uma variável são determinadas pelo seu tipo. Por exemplo, uma variável do tipo booleano não pode ser usada para armazenar valores de ponto flutuante.
Além disso, o tipo de uma variável não pode mudar durante a sua vida útil. Uma variável int não pode se transformar em uma variável char, por exemplo.

Todas as variáveis em Java devem ser declaradas antes de seu uso. Isso é necessário porque o compilador deve saber que tipo de dados uma variável contém antes de poder compilar corretamente qualquer instrução que use a variável. Ele também permite que o Java execute uma verificação de tipo rigorosa.

Inicializando uma variável

Em geral, você deve dar um valor a uma variável antes de usá-la. Uma maneira de dar um valor a uma variável é através de uma instrução de atribuição, como você já viu.

Outra maneira é dando-lhe um valor inicial quando é declarado. Para fazer isso, siga o nome da variável com um sinal de igual e o valor que está sendo atribuído.

A forma geral de inicialização é mostrada aqui: tipo var = valor;

Aqui, valor é o valor que é dado a var quando var é criado. O valor deve ser compatível com o tipo especificado. Eis alguns exemplos:

```
int count = 10; // give count an initial value of 10
char ch = 'X'; // initialize ch with the letter X
float f = 1.2F; // f is initialized with 1.2
```

Inicializando uma variável

Ao declarar duas ou mais variáveis do mesmo tipo usando uma lista separada por vírgula, você pode dar a uma ou mais dessas variáveis um valor inicial. Por exemplo:

```
int a, b = 8, c = 19, d; // b and c have initializations
```

Neste caso, apenas b e c são inicializados.

Inicialização dinâmica

Embora os exemplos anteriores tenham usado apenas constantes como inicializadores, o Java permite que as variáveis sejam inicializadas dinamicamente, usando qualquer expressão válida no momento em que a variável é declarada.

Por exemplo, aqui está um pequeno programa que calcula o volume de um cilindro dado o raio de sua base e sua altura:

```
// Demonstrate dynamic initialization.
class DynInit {
    public static void main(String args[]) {
      double radius = 4, height = 5;

      // dynamically initialize volume
      double volume = 3.1416 * radius * radius * height;

      System.out.println("Volume is " + volume);
    }
}
```

volume is dynamically initialized at run time.

Aqui, três variáveis locais — raio, altura e volume — são declaradas. Os dois primeiros, raio e altura, são inicializados por constantes. No entanto, o volume é inicializado dinamicamente para o volume do cilindro. O ponto-chave aqui é que a expressão de inicialização pode usar qualquer elemento válido no momento da inicialização, incluindo chamadas para métodos, outras variáveis ou literais.

O âmbito e o tempo de vida das variáveis

Até agora, todas as variáveis que temos usado foram declaradas no início do método main( ). No entanto, o Java permite que as variáveis sejam declaradas dentro de qualquer bloco.

Um bloco é iniciado com uma cinta de abertura e terminado por uma cinta de fechamento. Um bloco define um escopo.

Assim, cada vez que você inicia um novo bloco, você está criando um novo escopo. Um escopo determina quais objetos são visíveis para outras partes do programa. Também determina o tempo de vida desses objetos.

Em geral, cada declaração em Java tem um escopo. Como resultado, Java define um conceito poderoso e refinado de escopo. Dois dos escopos mais comuns em Java são aqueles definidos por uma classe e aqueles definidos por um método.

O âmbito e o tempo de vida das variáveis

O escopo definido por um método começa com sua chave de abertura. No entanto, se esse método tiver parâmetros, eles também são incluídos no escopo do método. O escopo de um método termina com sua chave de fechamento. Este bloco de código é chamado de corpo do método.

Como regra geral, as variáveis declaradas dentro de um escopo não são visíveis (ou seja, acessíveis) para o código definido fora desse escopo. Assim, quando você declara uma variável dentro de um escopo, você está localizando essa variável e protegendo-a contra acesso e/ou modificação não autorizados. Com efeito, as regras de âmbito de aplicação fornecem a base para o encapsulamento. Uma variável declarada dentro de um bloco é chamada de variável local.

Os escopos podem ser aninhados. Por exemplo, cada vez que você cria um bloco de código, você está criando um novo escopo aninhado. Quando isso ocorre, o escopo externo encerra o escopo interno. Isso significa que os objetos declarados no escopo externo serão visíveis para o código dentro do escopo interno. No entanto, o inverso não é verdade. Os objetos declarados dentro do escopo interno não serão visíveis fora dele.

Para entender o efeito dos escopos aninhados, considere o seguinte programa:

O âmbito e o tempo de vida das variáveis

```
// Demonstrate block scope.
class ScopeDemo {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope

      int y = 20; // known only to this block

      // x and y both known here.

      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here <-- Here, y is outside of its scope.

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

Entradas e saídas de dados através da consola

Para tornar nossos programas de exemplo mais interessantes, queremos aceitar a entrada e formatar adequadamente a saída do programa. Obviamente, os programas modernos usam uma GUI para coletar entradas do utilizador.

No entanto, programar tal interface requer mais ferramentas e técnicas do que temos à nossa disposição no momento.

Nossa primeira tarefa é nos familiarizarmos com a linguagem de programação Java, então usamos o humilde console para entrada e saída.

Entrada de dados

Você viu que é fácil imprimir a saída no “fluxo de saída padrão” (ou seja, a janela do console) apenas chamando System.out.println. A leitura do "fluxo de entrada padrão" System.in não é tão simples. Para ler a entrada do console, você primeiro constrói um Scanner que está anexado ao System.in:

```
Scanner in = new Scanner(System.in);
```

Entrada de dados

Agora podemos usar os vários métodos da classe Scanner para ler a entrada. Por exemplo, o método nextLine lê uma linha de entrada.

```
System.out.print("What is your name? ");
String name = in.nextLine();
```

Aqui, usamos o método nextLine porque a entrada pode conter espaços.
Para ler uma única palavra (delimitada por espaço em branco), chame

```
String firstName = in.next();
```

Para ler um número inteiro, utilize o método nextInt.

```
System.out.print("How old are you? ");
int age = in.nextInt();
```

Da mesma forma, o método nextDouble lê o próximo número de ponto flutuante.

Observação

A classe Scanner não é adequada para ler uma senha de um console, pois a entrada é claramente visível para qualquer pessoa.

Use a classe Console para essa finalidade. Para ler uma senha, use o seguinte código:

```
Console cons = System.console();
String username = cons.readLine("User name: ");
char[] passwd = cons.readPassword("Password: ");
```

THANK YOU!

CONTACT US AT:

Nelson Norte

nnorte@ucm.ac.mz

828848766