CS6308- Java Programming

V P Jayachitra
Assistant Professor
Department of Computer Technology
MIT Campus
Anna University

Java's fundamental elements

- Data types
- Variables
- Arrays

Data types

- **No automatic coercions or conversions of conflicting types.**
- **Primitive data type**
- **Non primitive type**

Primitive data type

- Java defines eight primitive types of data: **byte, short, int, long, char, float, double, and boolean**

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

Primitive data type contd…

- Java defines eight primitive types of data: **byte, short, int, long, char, float, double, and boolean**

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038
char	16	0 to 65,536
boolean	-	true or false

Integer literals

- When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type.
  - An integer literal can always be assigned to a long variable.
  - long literal ?
    - you will need to explicitly tell the compiler that the literal value is of type long.
    - Do this by appending an upper- or lowercase L to the literal.
      - For example, 0x7fffffffffffffL or 9223372036854775807L is the largest long.
  - An integer can also be assigned to a char as long as it is within range.
  - Decimal numbers cannot have a leading zero. Therefore signify a hexadecimal constant with a leading zero-x, (0x or 0X).
    - int x = 0b1010; //0B or 0b for binary
  - embed one or more underscores in an integer literal. cannot come at the beginning or the end of a literal
    - int x = 123_456_789; // x will be 123456789. The underscores will be ignored.
    - int x = 123___456___789; // x will be 123456789. The underscores will be ignored.
    - int x = 0b1101_0101_0001_1010; // x will be 54554

Float literal

- **a float literal, must append an *F* or *f* to the constant.**
- **A double literal, must append a *D* or *d to the constant.***
- ***When the literal is compiled, the underscores are discarded.***
- **double num = 9_423_497.1_0_9; // 9423497.109.**
- **double num = 9_423_497_862.0;**

Boolean Literals

- **Two logical values that a boolean value can have, true and false.**
- **The values of true and false do not convert into any numerical representation.**
- **The true literal in Java does not equal 1, nor does the false literal equal 0.**

Character Literals

- A literal character is represented inside a pair of single quotes.
- All of the visible ASCII characters can be directly entered inside the quotes, such as '*a*', '*z*', and '@'.
- For characters that are impossible to enter directly, use escape sequences that allow you to enter the character
  - ' \' ' for the single-quote character itself and ' \n' for the newline character.
- For octal notation, use the backslash followed by the three-digit number.
  - For example, ' *141*' is the letter '*a*'.
- enter a backslash-u (\u), then exactly four hexadecimal digits.
  - For hexadecimal example, ' *u0061*' is the ISO-Latin-1 '*a*'
  - ' *ua432* ' is a Japanese Katakana character.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

String Literals

- **String literals in Java are specified by enclosing a sequence of characters between a pair of double quotes.**
- **Examples of string literals are**

```
"Hello World"
"two\nlines"
" \"This is in quotes\""
```

Variables

- **The variable is the basic unit of storage in a Java program.**
- **A variable is defined by the combination of an identifier, a type, and an optional initializer.**
- **In addition, all variables have a scope, which defines their visibility, and a lifetime.**

Declaring a Variable

- **In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:**

type identifier [= *value*][, *identifier* [= *value*] …];

Variables contd…

Variable declarations of various types.

```
int a, b, c;              // declares three ints, a, b, and c.
int d = 3, e, f = 5;      // declares three more ints, initializing
                          // d and f.
byte z = 22;              // initializes z.
double pi = 3.14159;      // declares an approximation of pi.
char x = 'x';             // the variable x has the value 'x'.
```

Variables contd…

Dynamic Initialization

hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
class DynInit {
  public static void main(String args[]) {
    double a = 3.0, b = 4.0;

    // c is dynamically initialized
    double c = Math.sqrt(a * a + b * b);

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

Scope

```
// Demonstrate block scope.
class Scope {
  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

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

Scope

```
// This fragment is wrong!
count = 100; // oops! cannot use count before it is declared!
int count;
```

```
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y is: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

Scope

- Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

```
// This program will not compile
class ScopeErr {
  public static void main(String args[]) {
    int bar = 1;
    {                  // creates a new scope
      int bar = 2;  // Compile-time error - bar already defined!
    }
  }
}
```

Type Conversion and Casting

- If the two types are compatible, then Java will perform the conversion automatically.
  - For example, it is always possible to assign an int value to a long variable.
  - For instance, there is no automatic conversion defined from double to byte.
    - use a *cast*, which performs an explicit conversion between incompatible types.
- Java's Automatic Conversions: no explicit cast statement is required.
  - The two types are compatible.
  - The destination type is larger than the source type.
  - no automatic conversions from the numeric types to char or boolean.
  - performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types

- what if you want to assign an int value to a byte variable?
- **This conversion will not be performed automatically, because a byte is smaller than an int.**
- **This kind of conversion is sometimes called a narrowing conversion**
  - **explicitly making the value narrower so that it will fit into the target type.**
- **To create a conversion between two incompatible types, you must use a cast.**
- **A cast is simply an explicit type conversion. It has this general form:**
  - **(target-type) value**

```
int a;
byte b;
// …
b = (byte) a;
```

Casting Incompatible Types

```
// Demonstrate casts.
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nConversion of int to byte.");
    b = (byte) i;
    System.out.println("i and b " + i + " " + b);

    System.out.println("\nConversion of double to int.");
    i = (int) d;
    System.out.println("d and i " + d + " " + i);

    System.out.println("\nConversion of double to byte.");
    b = (byte) d;
    System.out.println("d and b " + d + " " + b);
  }
}
```

Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67

- **Automatic Type Promotion in Expressions**

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

```
class Promote {
  public static void main(String args[]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
  }
}
```

- **Type Promotion Rules**
  - First, all byte, short, and char values are promoted to int, as just described.
  - Then, if one operand is a long, the whole expression is promoted to long.
  - If one operand is a float, the entire expression is promoted to float.
  - If any of the operands are double, the result is double.

Arrays

- **An array is a group of like-typed variables that are referred to by a common name.**
- **Arrays of any type can be created and may have one or more dimensions.**
- **A specific element in an array is accessed by its index.**

One-Dimensional Arrays

type var-name[ ];
array-var = new type [size];

int month_days[];
month_days = new int[12];

int month_days[] = new int[12];

```
// An improved version of the previous program.
class AutoArray {
  public static void main(String args[]) {

    int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                         30, 31 };
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

```
// Average an array of values.
class Average {
  public static void main(String args[]) {
    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;
    for(i=0; i<5; i++)
      result = result + nums[i];
    System.out.println("Average is " + result / 5);
  }
}
```

Multidimensional Arrays

int twoD[][] = new int[4][5];

```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
      twoD[i][j] = k;
      k++;
    }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

When you run this program, you will get the following output:

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

Alternative Array Declaration Syntax

```
type[ ] var-name;
```

```
int al[] = new int[3];
int[] a2 = new int[3];
```

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

```
int[] nums, nums2, nums3; // create three arrays
int nums[], nums2[], nums3[]; // create three arrays
```

Introducing Type Inference with Local Variables

- **Recently, an exciting new feature called *local variable type inference* was added to the Java language.**
- **In the past, all variables required an explicitly declared type, whether they were initialized or not.**
- **Beginning with JDK 10, it is now possible to let the compiler infer the type of a local variable based on the type of its initializer, thus avoiding the need to explicitly specify the type.**
- **Local variable type inference offers a number of advantages.**
  - **eliminating the need to redundantly specify a variable's type when it can be inferred from its initializer.**
  - **It can simplify declarations in cases in which the type name is quite lengthy, such as can be the case with some class names.**

Introducing Type Inference with Local Variables

```
double avg = 10.0;
var avg = 10.0;
```

```
Value of avg: 10.0
Value of var: 1
Value of k: -1
```

```
var myArray = new int[10]; // This is valid.
var[] myArray = new int[10]; // Wrong
var myArray[] = new int[10]; // Wrong
var counter; // Wrong! Initializer required.
var myArray = new int[10]; // This is valid.
var myArray = new int[10]; // This is valid.
```

The following program puts the preceding discussion into action:

```
// A simple demonstration of local variable type inference.
class VarDemo {
  public static void main(String args[]) {

    // Use type inference to determine the type of the
    // variable named avg. In this case, double is inferred.
    var avg = 10.0;
    System.out.println("Value of avg: " + avg);

    // In the following context, var is not a predefined identifier.
    // It is simply a user-defined variable name.
    int var = 1;
    System.out.println("Value of var: " + var);

    // Interestingly, in the following sequence, var is used
    // as both the type of the declaration and as a variable name
    // in the initializer.
    var k = -var;
    System.out.println("Value of k: " + k);
  }
}
```

Strings

- **String, is not a primitive type**

```
String str = "this is a test";
System.out.println(str);
```

Round a number using format

```
public class Decimal {
    public static void main(String[] args) {
                double num = 1.234567;
                System.out.format("%.2f", num);
    }
     }
```

Formatted output in Java

```
class JavaFormat
{
  public static void main(String args[])
  {
    int x = 10;
    System.out.printf("Print integer: x = %d\n", x);

    // print it upto 2 decimal places
    System.out.printf("Formatted with precision: PI = %.2f\n", Math.PI);
     float n = 5.2f;
     // automatically appends zero to the rightmost part of decimal
    System.out.printf("Formatted to specific width: n = %.4f\n", n);
     n = 2324435.3f;

    // width of 20 characters
    System.out.printf("Formatted to right margin: n = %20.4f\n", n);
  }
}
```