# Chapter 3
# Data Types, Variables and Arrays

# JAVA IS A STRONGLY TYPED LANGUAGE

First, every variable has a type, every expression has a type, and every type is strictly defined.

Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.

# Primitive Data Types

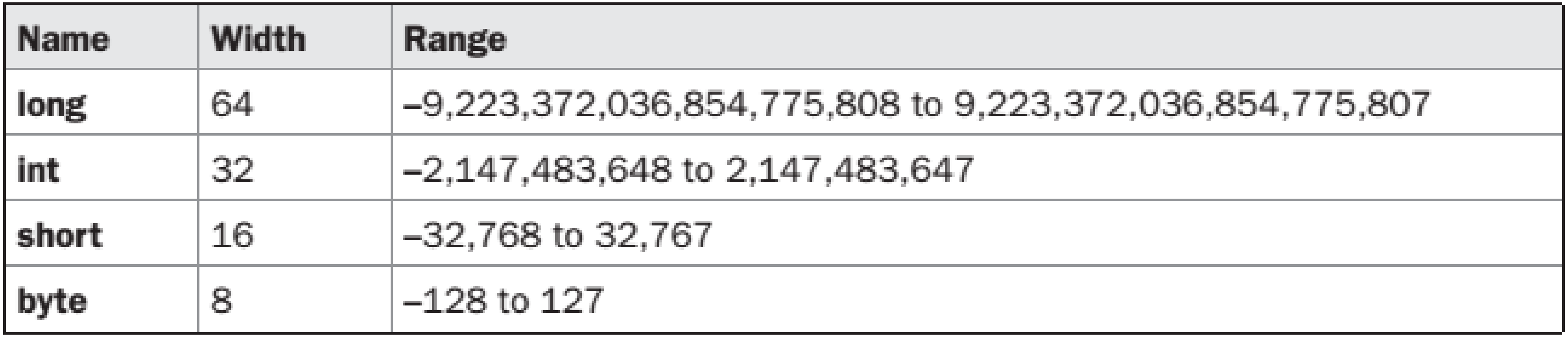
- Eight primitive types of data: byte, short, int, long, char, float, double, and boolean
- Put in four groups:
  - Integers: includes byte, short, int, and long
  - Floating-point numbers: includes float and double
  - Characters: includes char
  - Boolean: includes boolean
- Java is completely object oriented but primitive types are not

# Primitive Data Types

- C and C++ allow the size of an integer to vary based upon the dictates of the execution environment
- However, Java is different. Because of Java’s portability requirement, all data types have a strictly defined range for ex int is always 32 bits

# Integer

- All of these are signed, positive and negative values.
- ***Java does not support unsigned, positive-only integers***
- However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range for ex int is always 32 bits



| 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 |

# INTEGER → BYTE

- Variables of type **byte** are especially useful when you're working with a stream of data from a network or file.
- They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
- Ex: byte b, c;

# INTEGER → SHORT

- It is probably the least-used Java type.
- Ex.: short s;

# INTEGER → INT

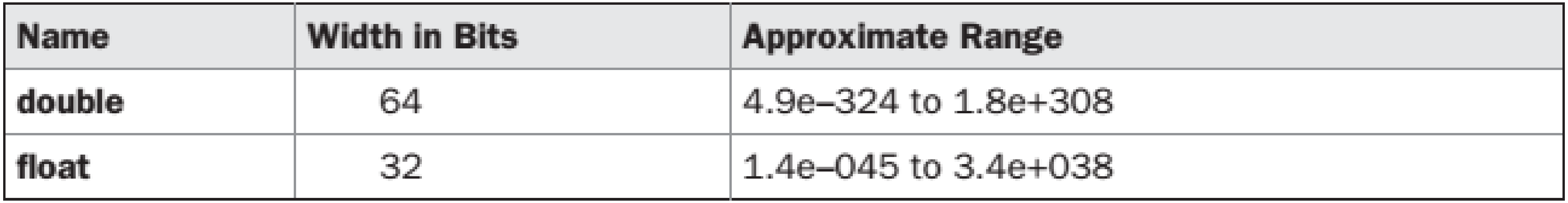
- most commonly used
- In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.
- **byte** and **short** values are used in an expression they are *promoted* to **int** when the expression is evaluated → ?
- Ex.: int s;

# INTEGER →LONG

- It is useful for those occasions where an int type is not large enough to hold the desired value.
- Ex: long days;

# FLOATING-POINT TYPES

- Also known as real numbers
- Two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively



| Name | Width in Bits | Approximate Range |
|---|---|---|
| **double** | 64 | 4.9e–324 to 1.8e+308 |
| **float** | 32 | 1.4e–045 to 3.4e+038 |

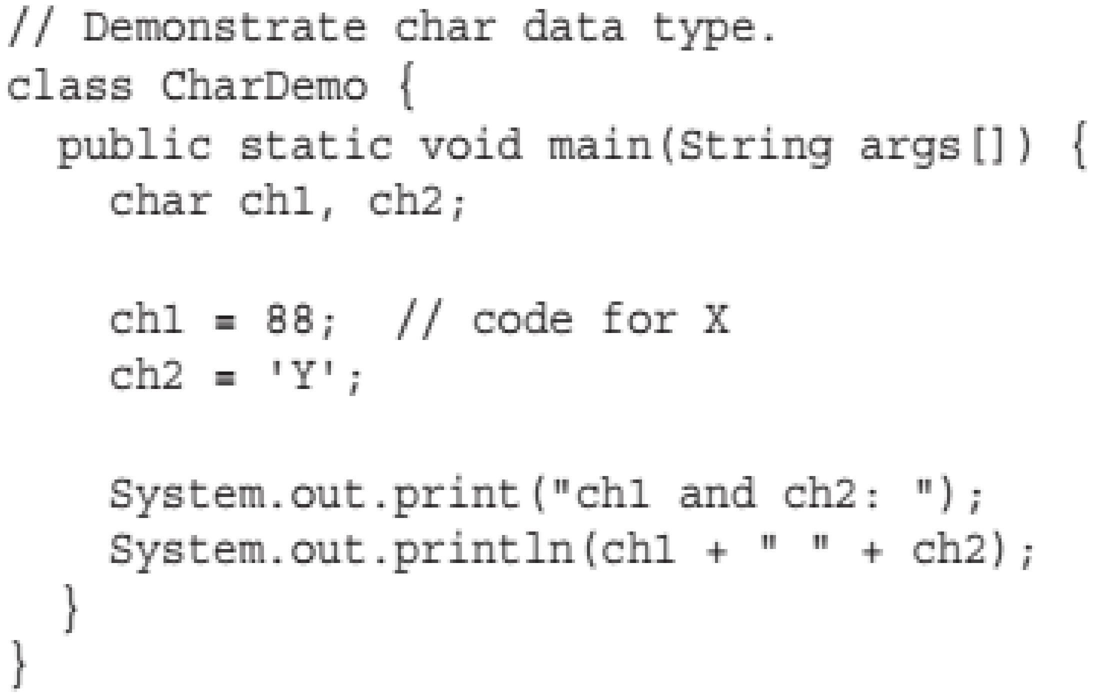
# FLOATING-POINT TYPES → FLOAT

- The type float specifies a single-precision value that uses 32 bits of storage. Single precision is
- faster on some processors and takes half as much space as double precision.
- Ex: float temp;

# FLOATING-POINT TYPES → DOUBLE

- Require more accuracy
- Ex: double pi=3.1416;

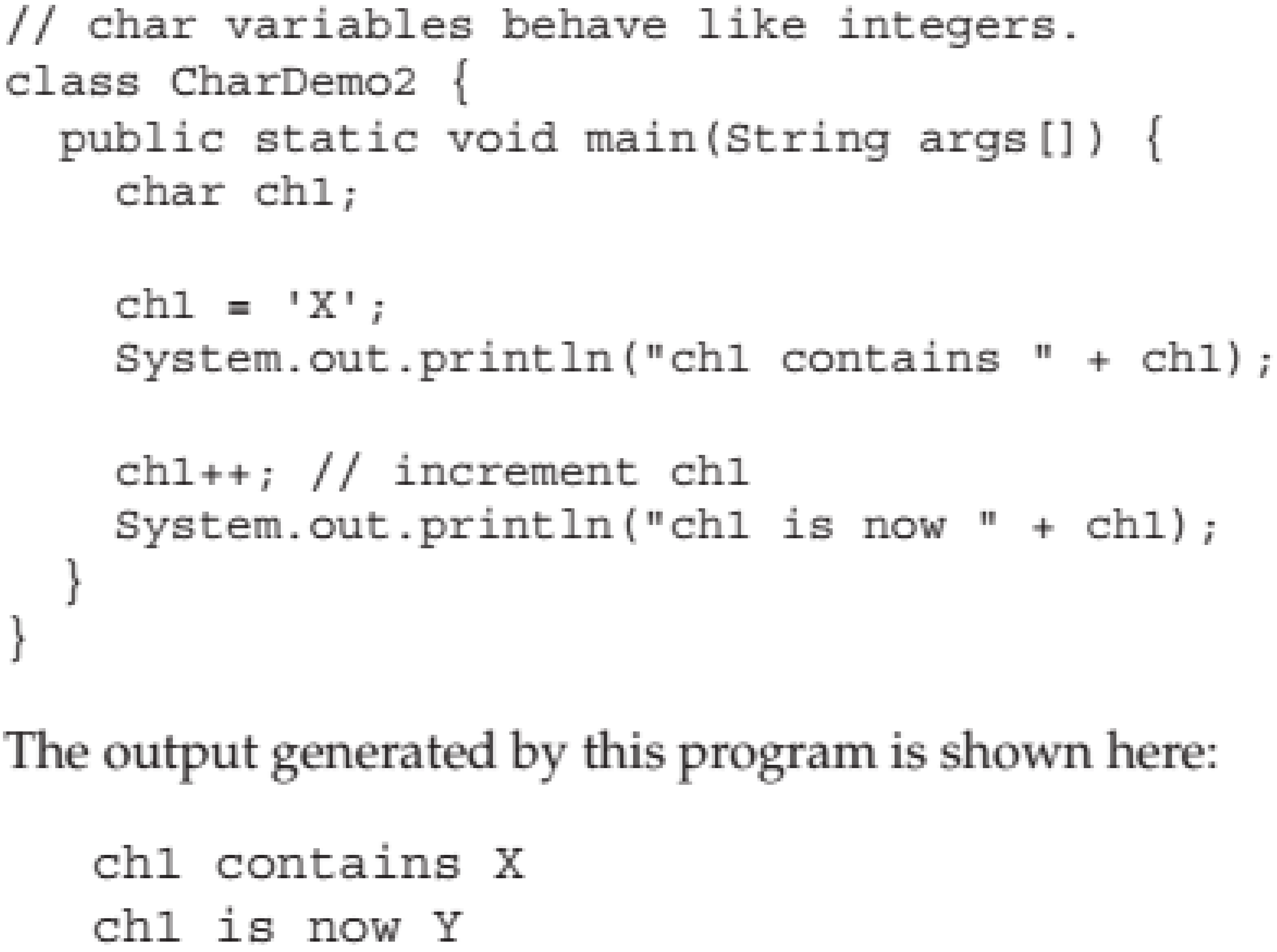
# CHARACTERS



```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88;  // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

- In C/C++, char is 8 bits wide
- Java uses ***Unicode to represent characters***
- Unicode defines a fully international character set that can represent all of the characters found in all human languages
- Java char is a 16-bit type
- No negative char. Range from 0 to 65536

# CHARACTERS



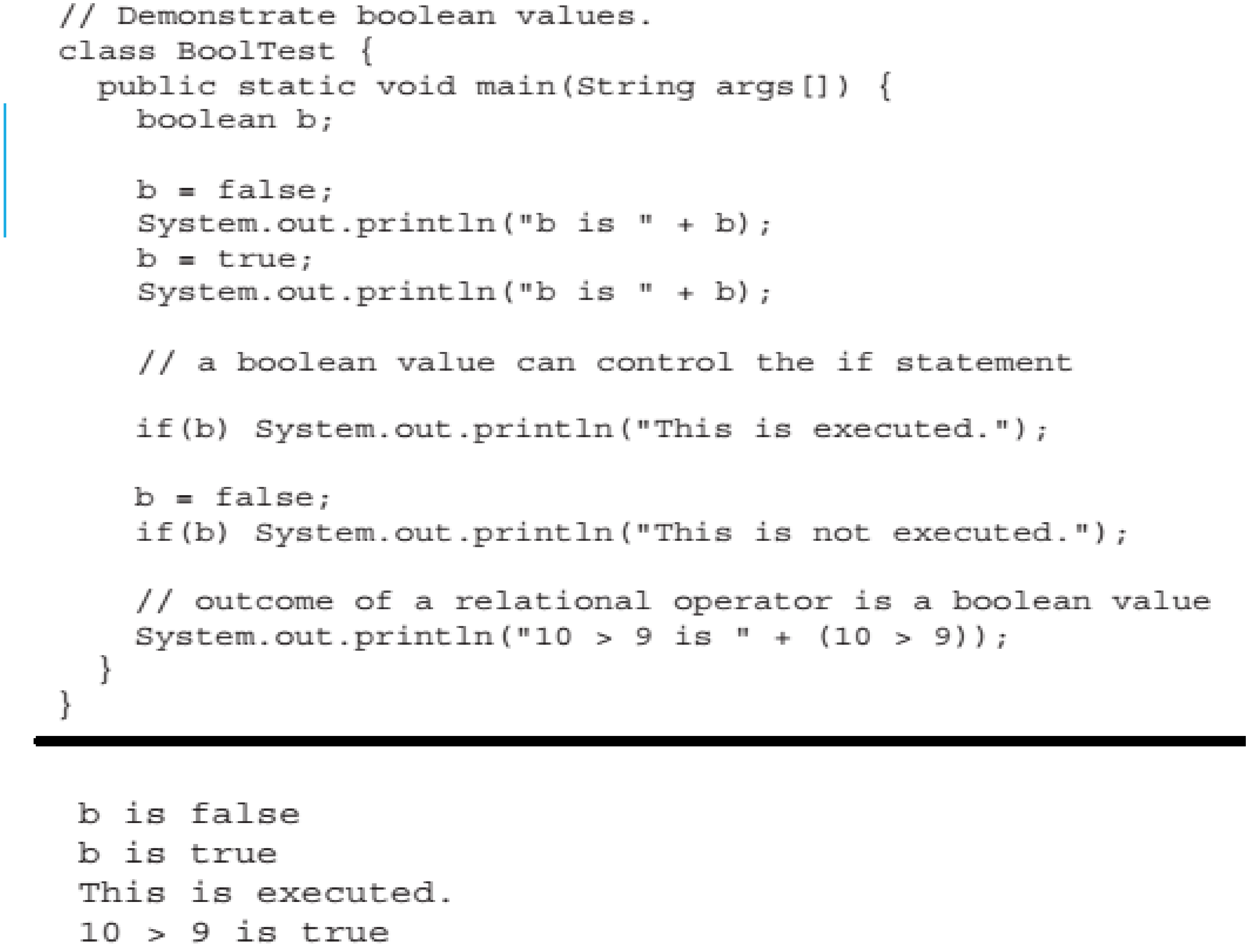
```
// char variables behave like integers.
class CharDemo2 {
  public static void main(String args[]) {
    char ch1;

    ch1 = 'X';
    System.out.println("ch1 contains " + ch1);

    ch1++; // increment ch1
    System.out.println("ch1 is now " + ch1);
  }
}
```

The output generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

# BOOLEANS

- Java has a primitive type, called **boolean**, for logical values.
- Two values, true or false
- This is the type returned by all relational operators, as in the case of **a < b**.



```
// Demonstrate boolean values.
class BoolTest {
  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));
  }
}
```

```
b is false
b is true
This is executed.
10 > 9 is true
```

# LITERALS

Integer literals : 1,2,3,5,7…,

- Octal – leading 0 eg 04
- Hex – leading 0x eg 0x10
- It is possible to assign an integer literal to one of Java's other integer types, such as byte or long as long as it is within range

Floating point literals :

- Standard notation – 2.0, 3.14234, 0.54334
- Scientific notation - 3.0544E12, 314234E-5
- Floating-point **literals** in Java default to **double** precision.

# LITERALS

Boolean literals :

- The values of true and false do not convert into any numerical representation

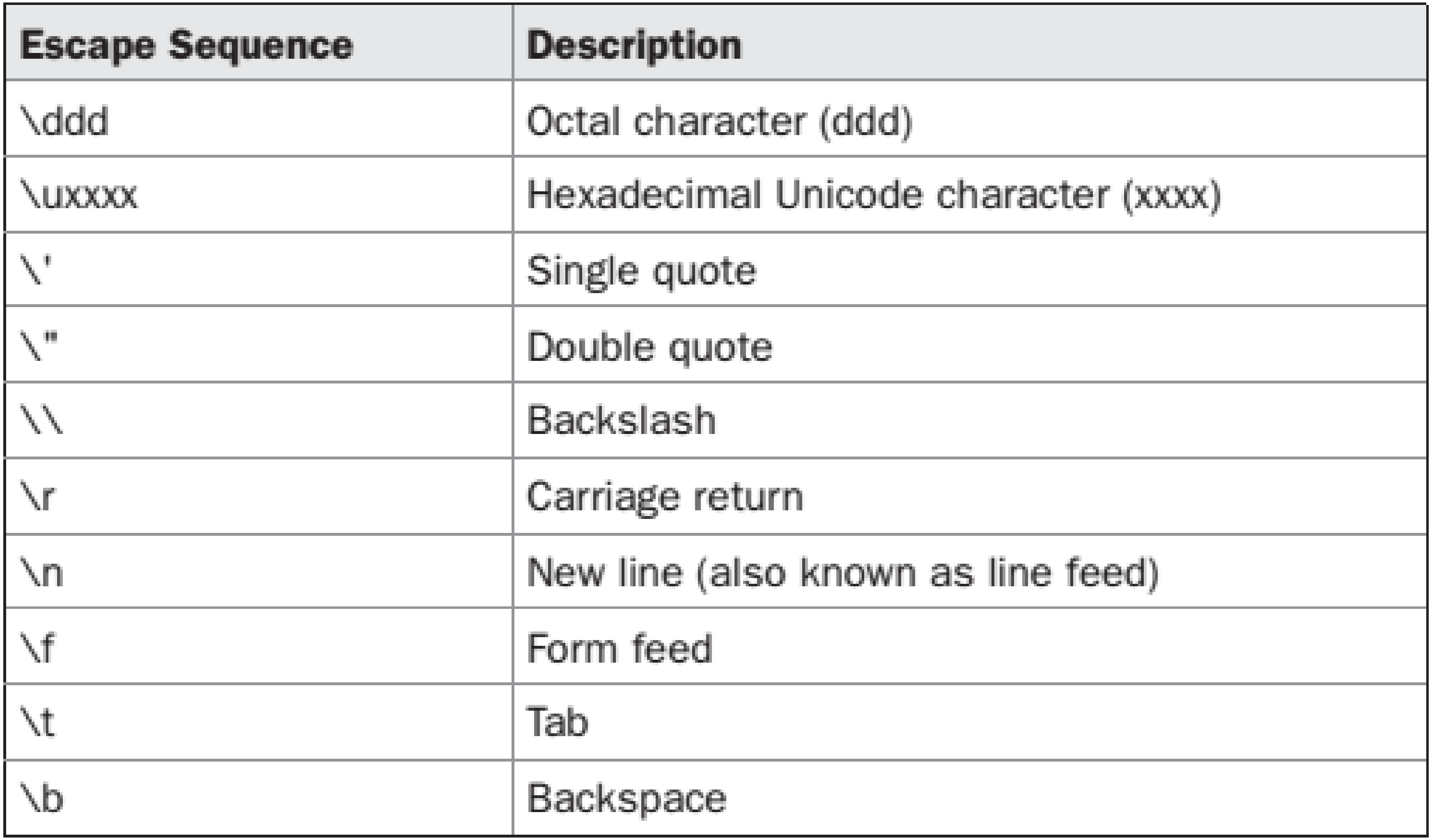
Character literals:

- Unicode characters enclosed in single quote. ‘X’, ‘\n’, ’\’, ‘\65’,
- For octal notation → ‘\3 digit’ → ‘\141’ = ‘a’
- For hexadecimal → ‘\ 4 hex digit’ → \u0061’ ISO-Latin-1 ‘a’

String literals:

- “Hello World” , “Double \n lines”, “\”In double quotes\””
- *One important thing to note about Java strings is that they must begin and end on the same line.*

# LITERALS

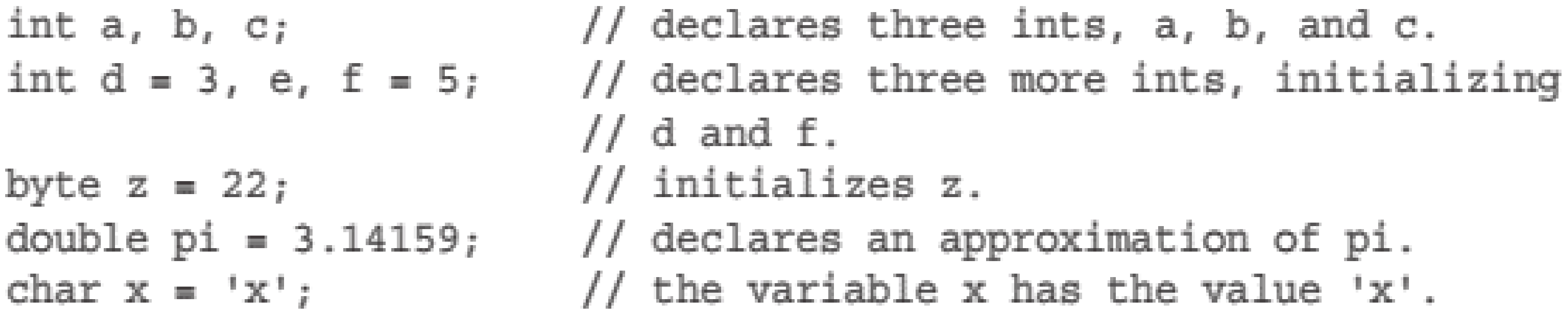


| Escape Sequence | Description |
|---|---|
| \ddd | Octal character (ddd) |
| \uxxxx | Hexadecimal Unicode character (xxxx) |
| \' | Single quote |
| \" | Double quote |
| \\ | Backslash |
| \r | Carriage return |
| \n | New line (also known as line feed) |
| \f | Form feed |
| \t | Tab |
| \b | Backspace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

# VARIABLES

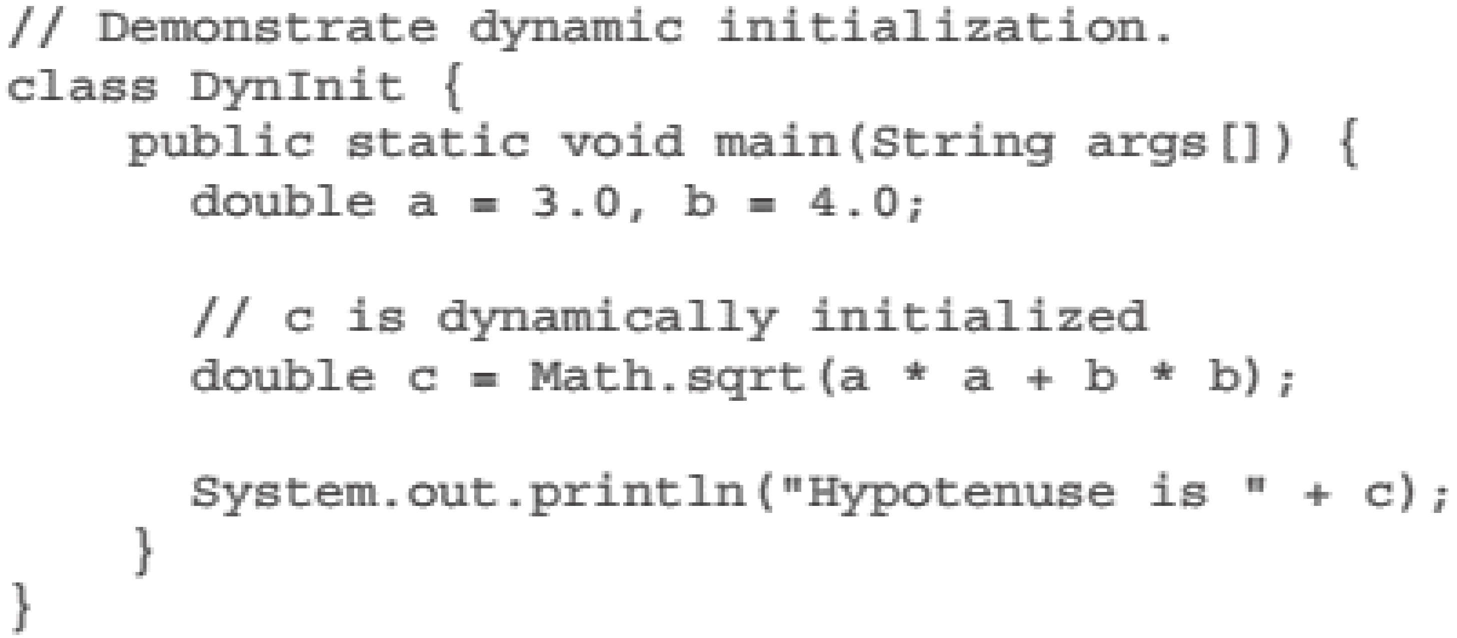
## Declaration

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



```
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'.
```

## Dynamic Initialization



```
// 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);
    }
}
```

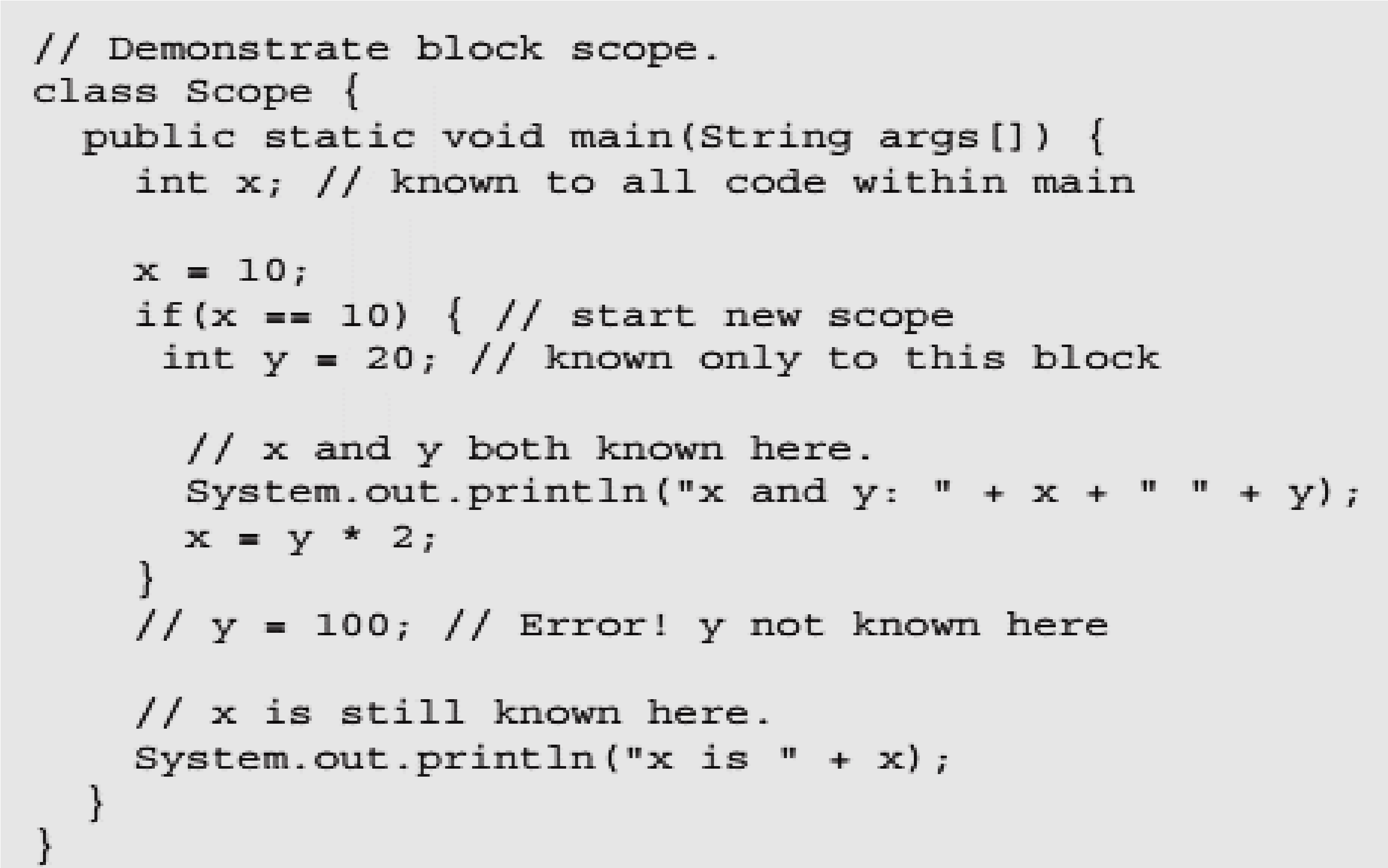
# VARIABLES

## Scope and lifetime of variable

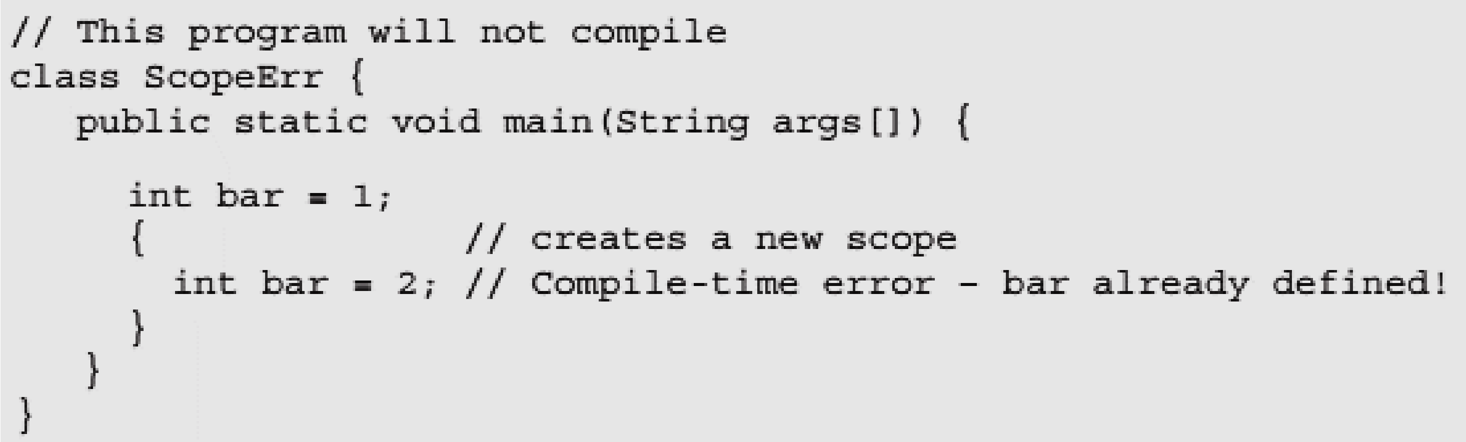
- A block defines the scope of a variable
- Java defines two types of scope; Class scope and method scope
- Class scope has unique properties and attributes that do not apply to method scope
- Variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope
- variables are created when their scope is entered, and destroyed when their scope is left
- Scopes can be nested
- Java does not permit using the same name again if a variable is declared in outer scope

# VARIABLES

Scope and lifetime of variable



```
// 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);
  }
}
```



```
// 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 CASTING AND CONVERSION

Automatic type conversion (/ Widening conversion) will take place if the following two conditions are met:

- 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

Explicit type casting → Narrowing conversion

- When a floating-point value is assigned to an integer type, the fractional component is lost
- If the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

# TYPE CASTING AND CONVERSION

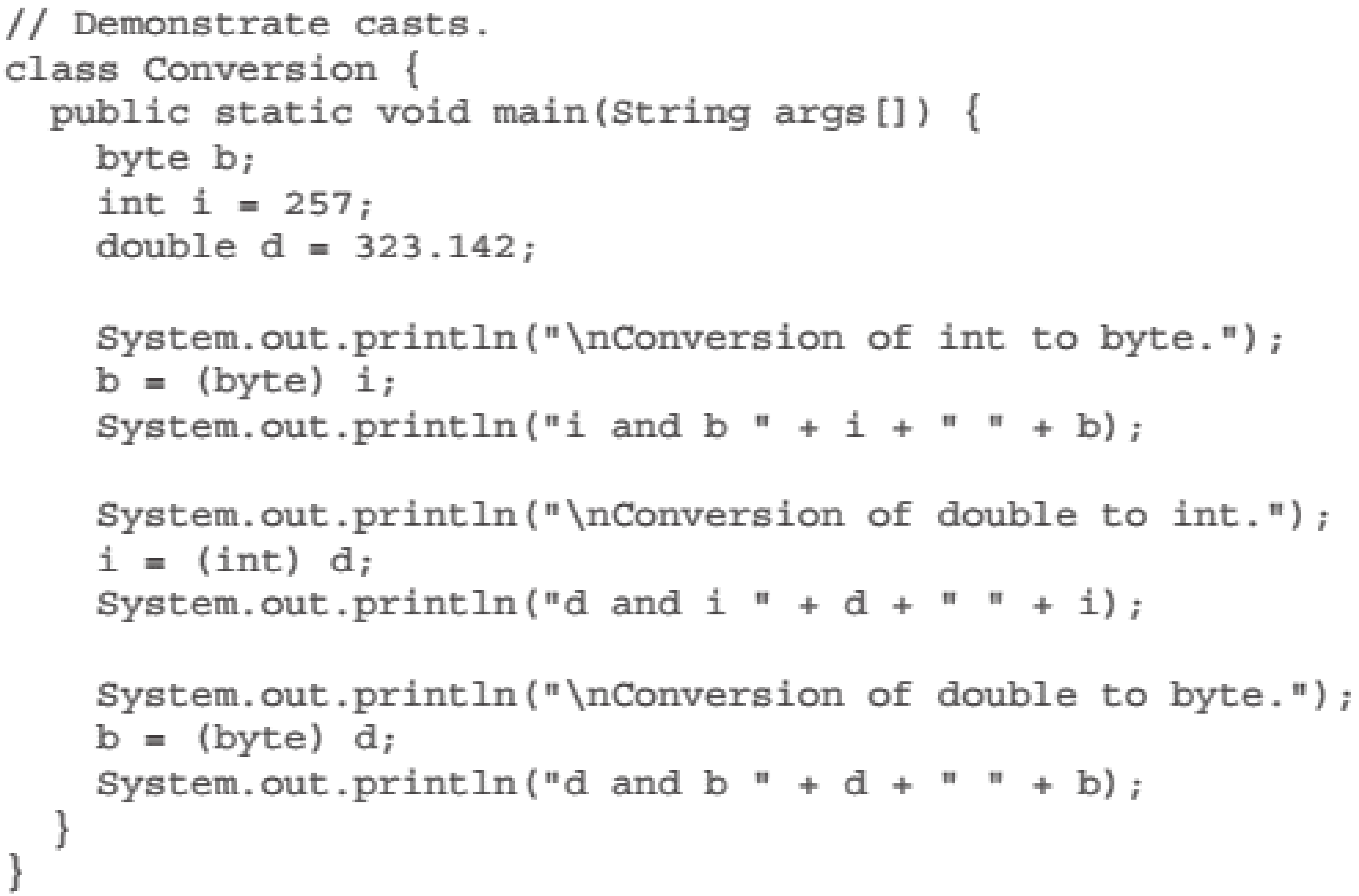
Explicit type casting → Narrowing conversion

- For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

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

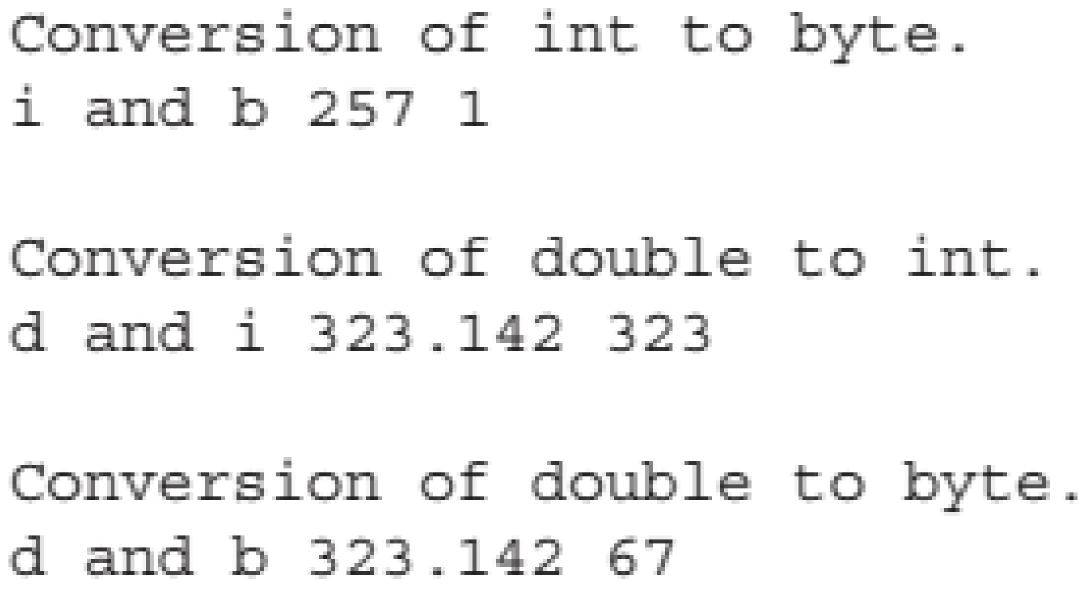
- A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*.

# TYPE CASTING AND CONVERSION



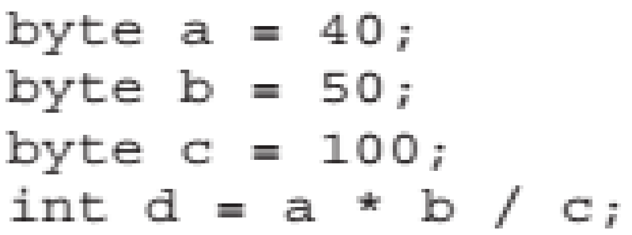
```
// Demonstrate casts.
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    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);
  }
}
```

# TYPE CASTING AND CONVERSION



```
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

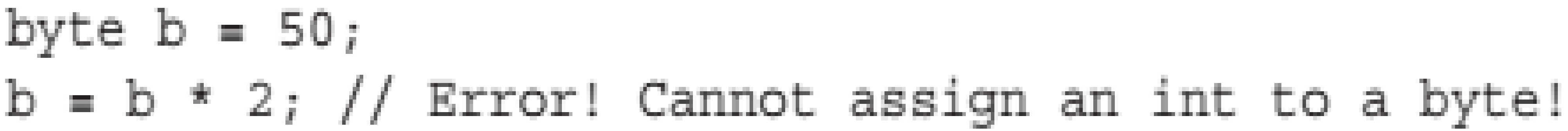


```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Here a * b exceeds byte range.

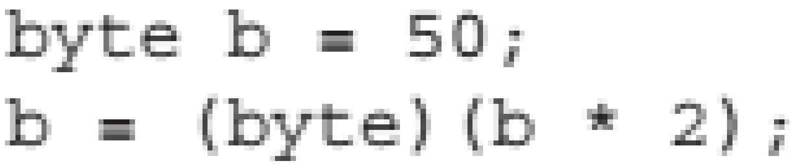
Java automatically promotes each byte, short, or char operand to int when evaluating an expression

Incorrect



```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

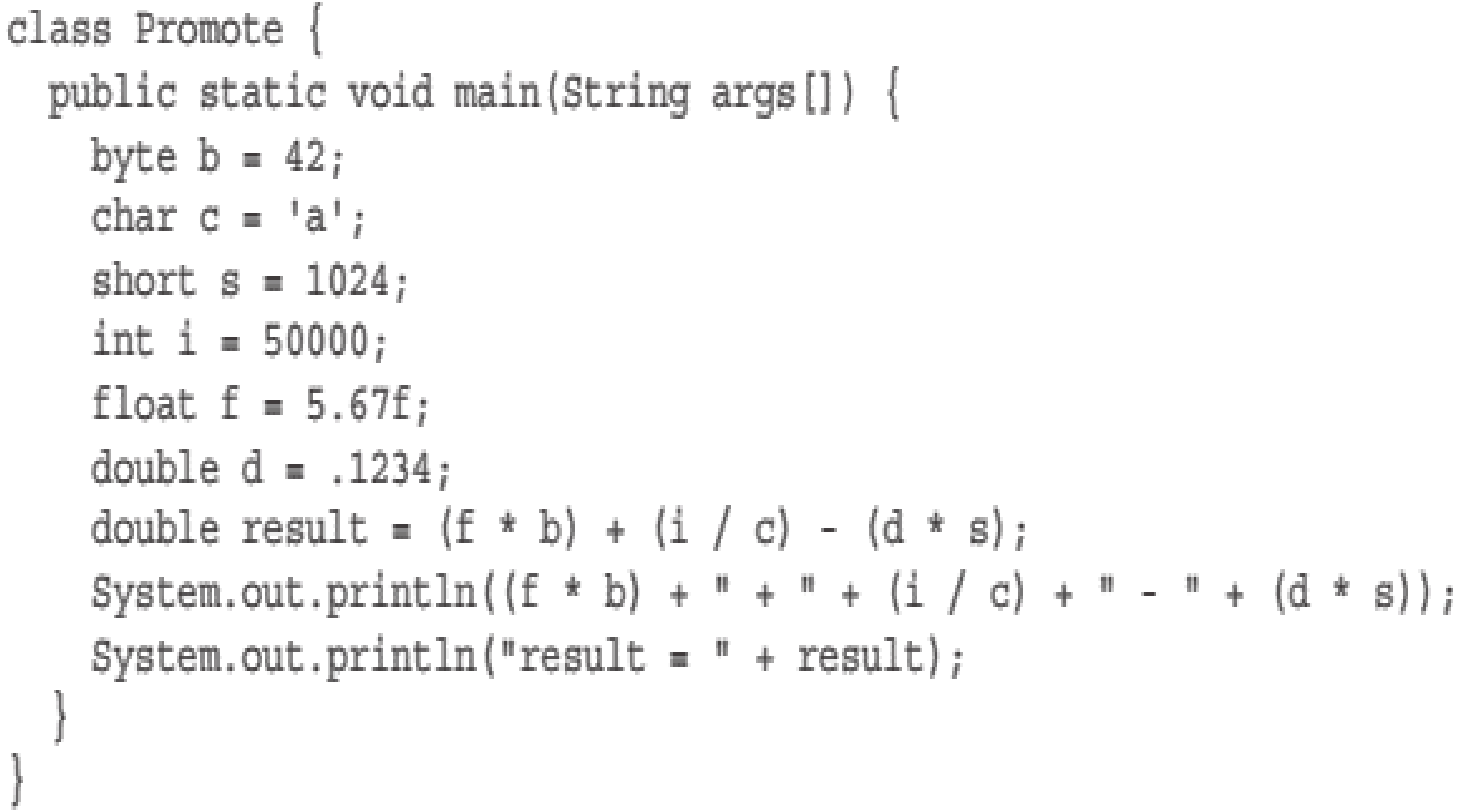
Correct



```
byte b = 50;
b = (byte)(b * 2);
```

# AUTOMATIC TYPE PROMOTION

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 is double, the result is double.



```
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);
  }
}
```

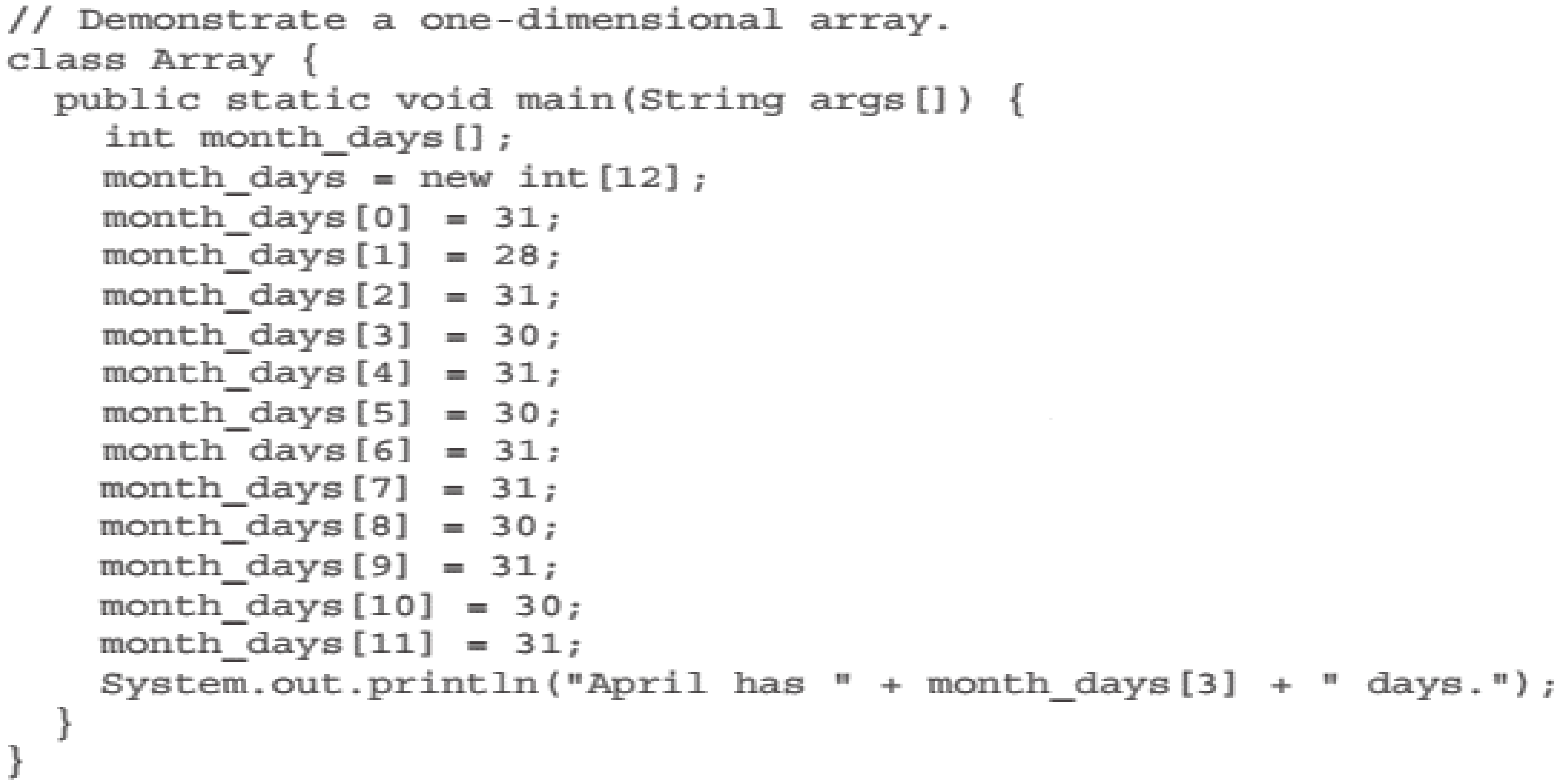
# ARRAYS

One-Dimensional Arrays

type varName[]; - declares an array

varName = new type[size] – allocates memory

Ex: month_day = new int[12]; - all elements initialized to zero

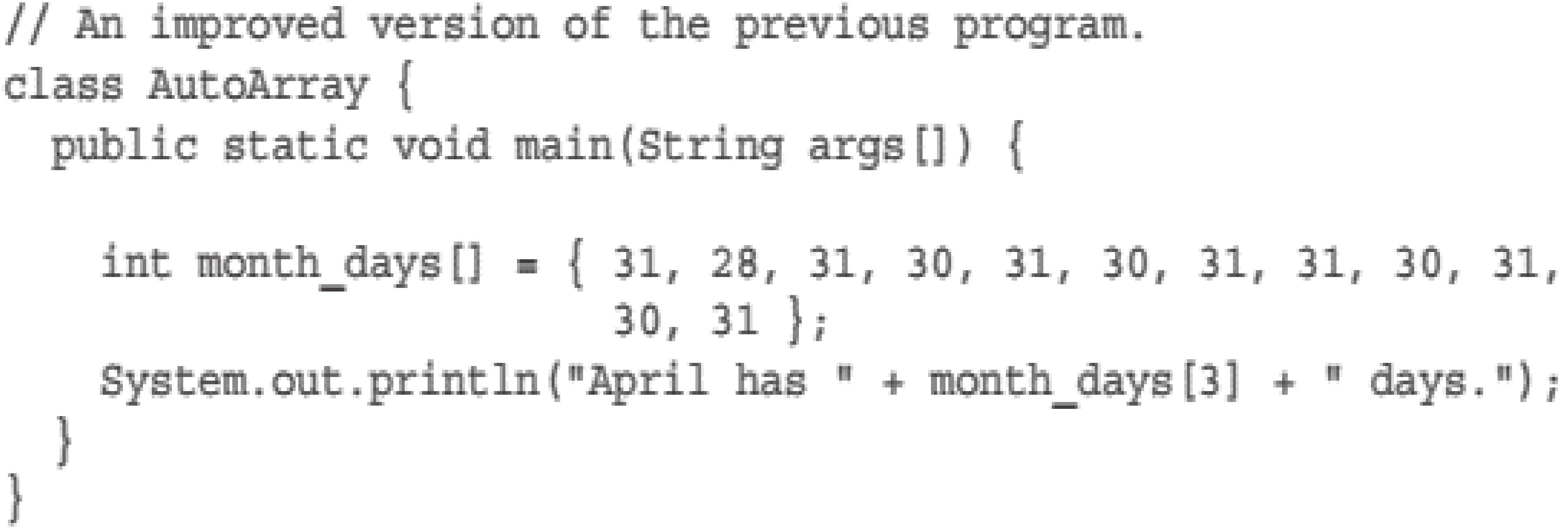


```
// Demonstrate a one-dimensional array.
class Array {
  public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

# ARRAYS

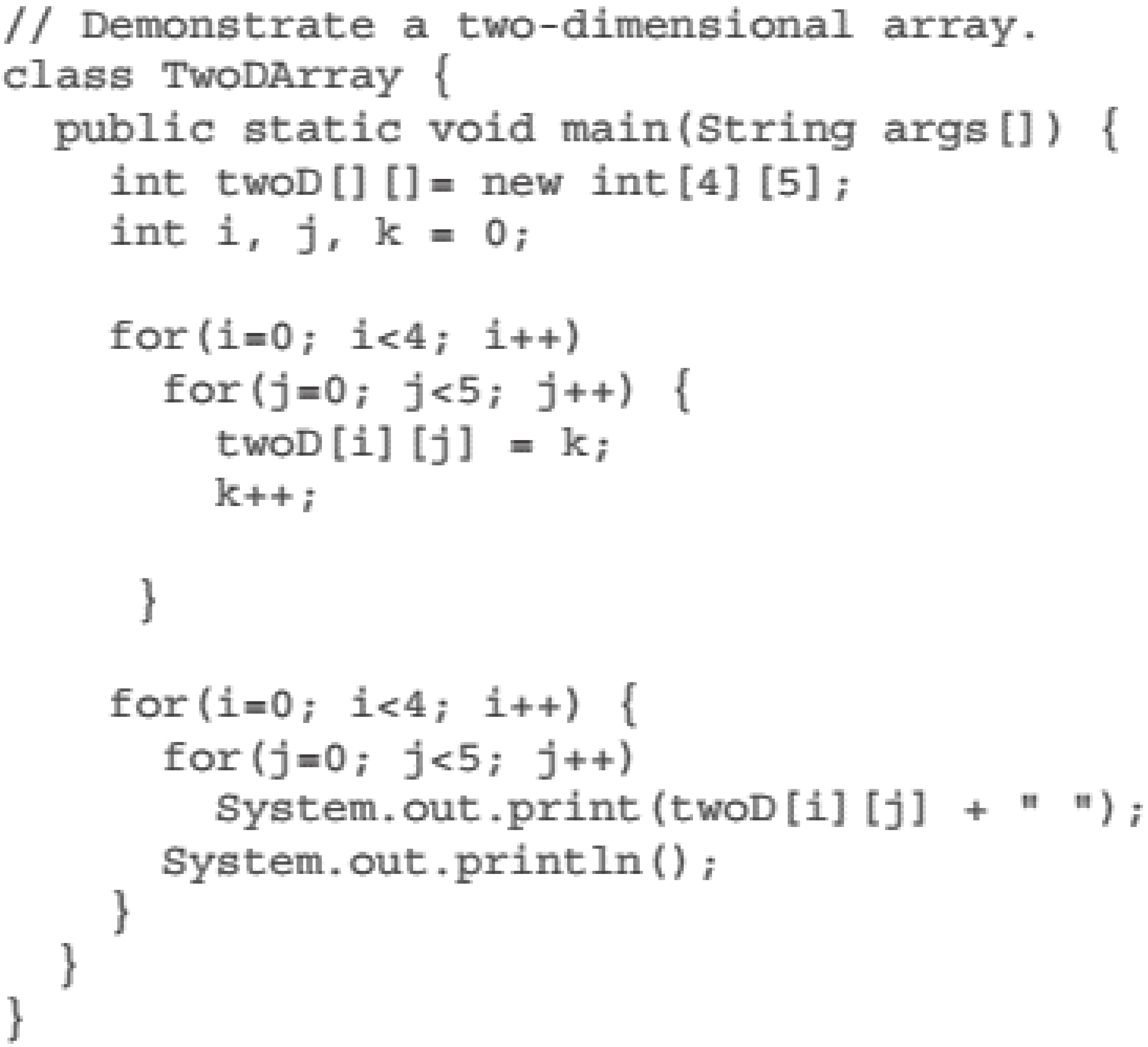
Java **strictly** checks to make sure you do not accidentally try to store or reference values outside of the range of the array.

An *array initializer* is a list of comma-separated expressions surrounded by curly braces.



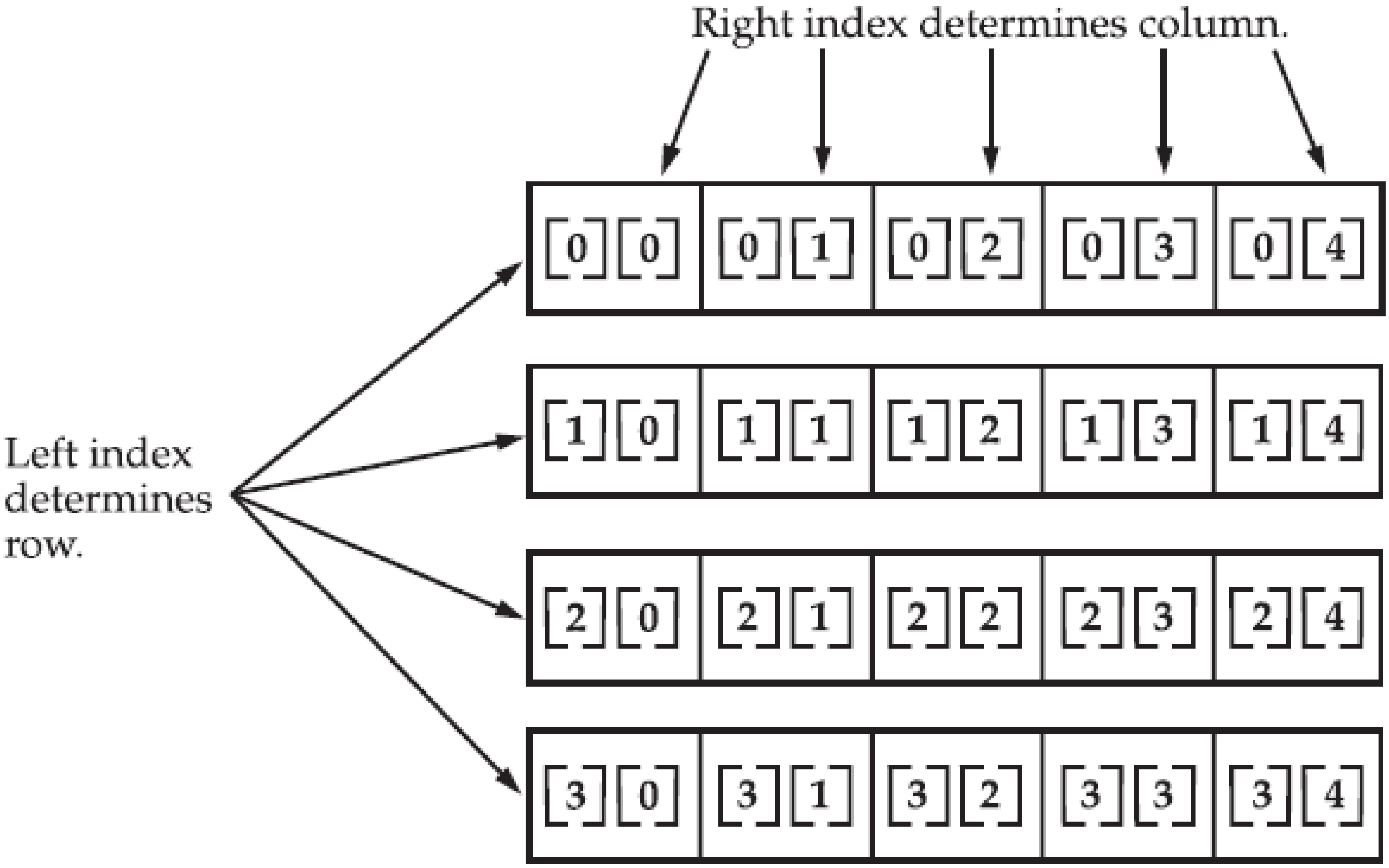
```
// An improved version of the previous program.
class AutoArray {
  public static void main(String args[]) {

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

# 2-D ARRAYS



```
// 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();
    }
  }
}
```

# 2-D ARRAYS



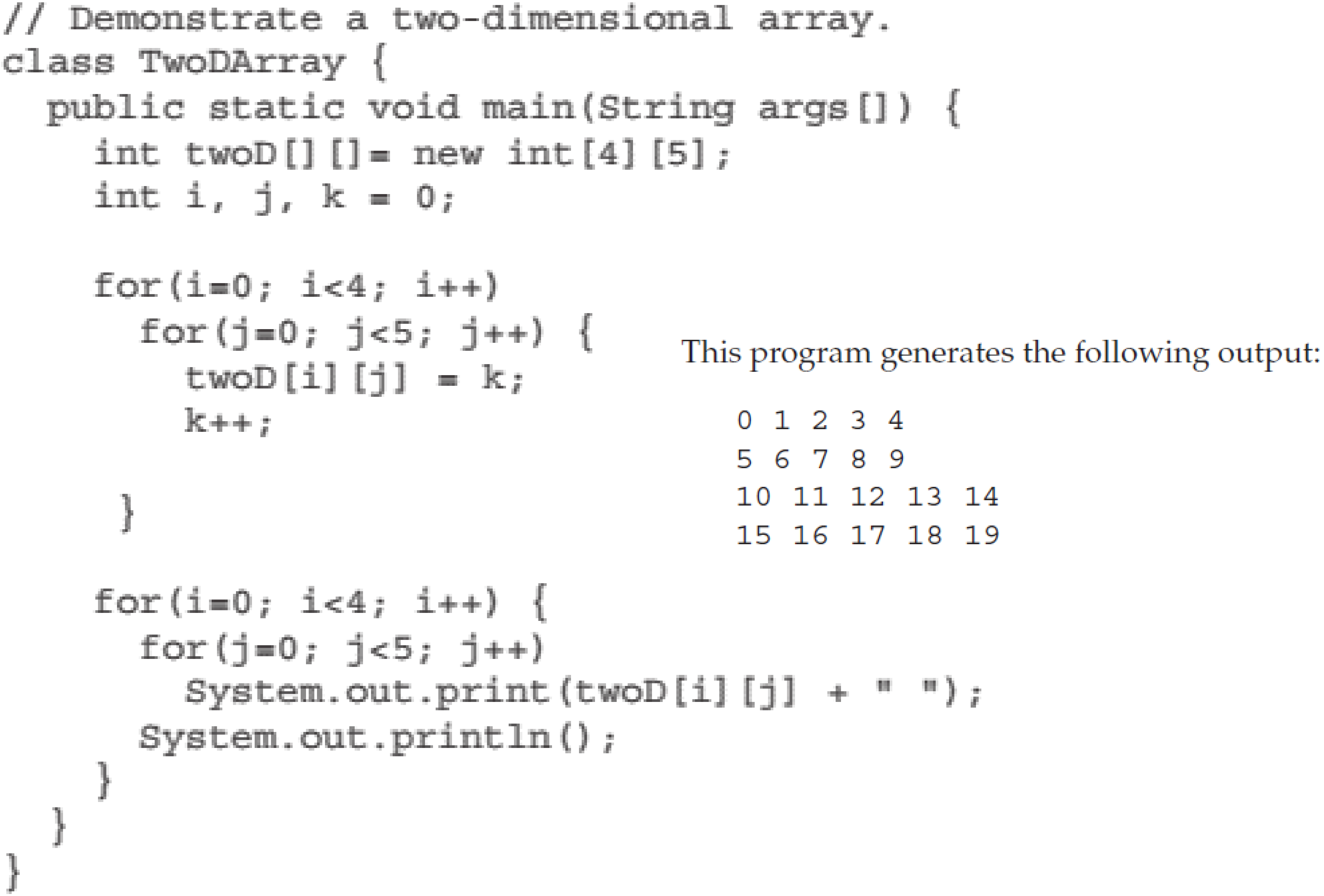
Right index determines column.

Left index determines row.

| | | | | |
|---|---|---|---|---|
| [0][0] | [0][1] | [0][2] | [0][3] | [0][4] |
| [1][0] | [1][1] | [1][2] | [1][3] | [1][4] |
| [2][0] | [2][1] | [2][2] | [2][3] | [2][4] |
| [3][0] | [3][1] | [3][2] | [3][3] | [3][4] |

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

# 2-D ARRAYS

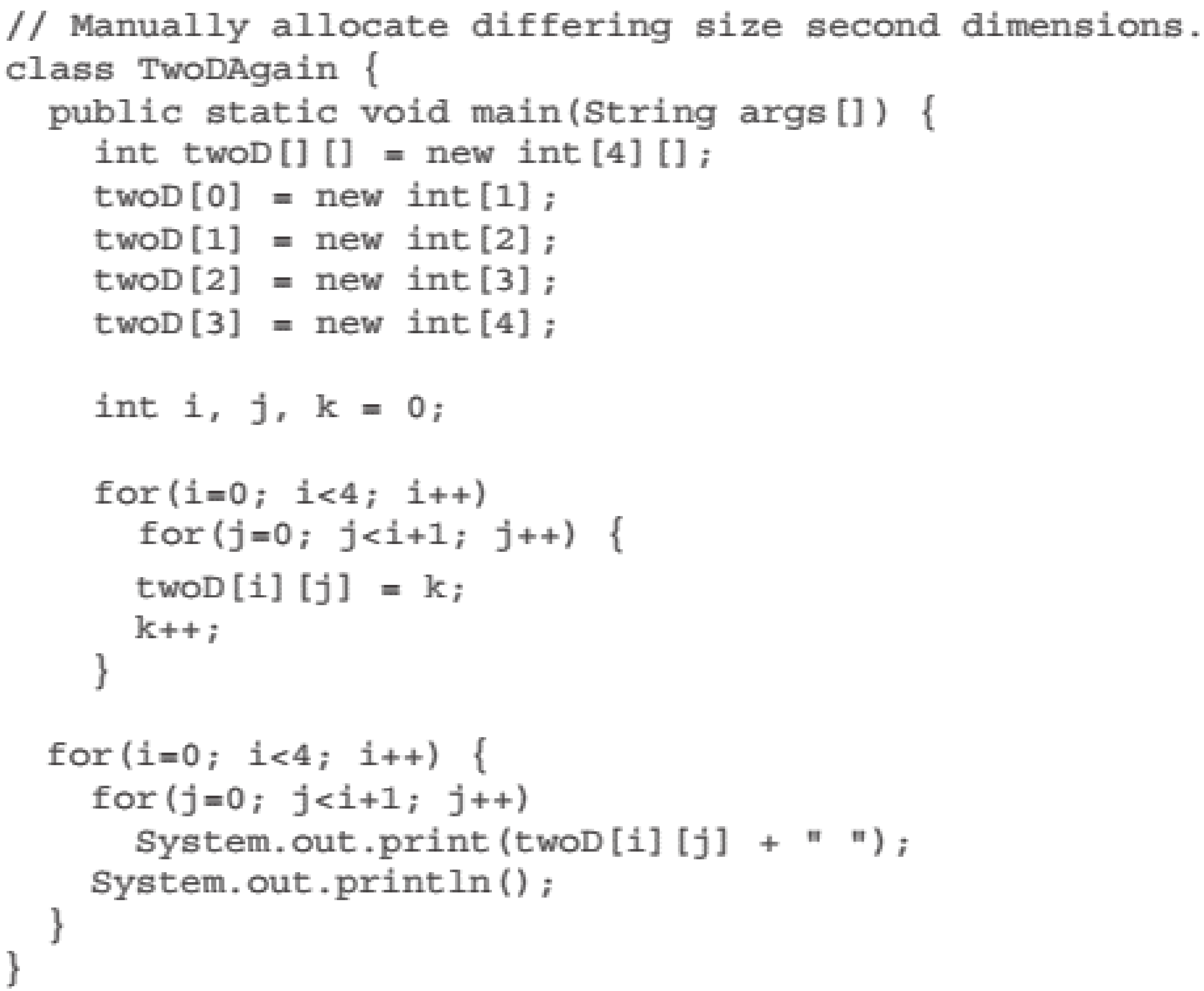


```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    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();
    }
  }
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

# 2-D ARRAYS



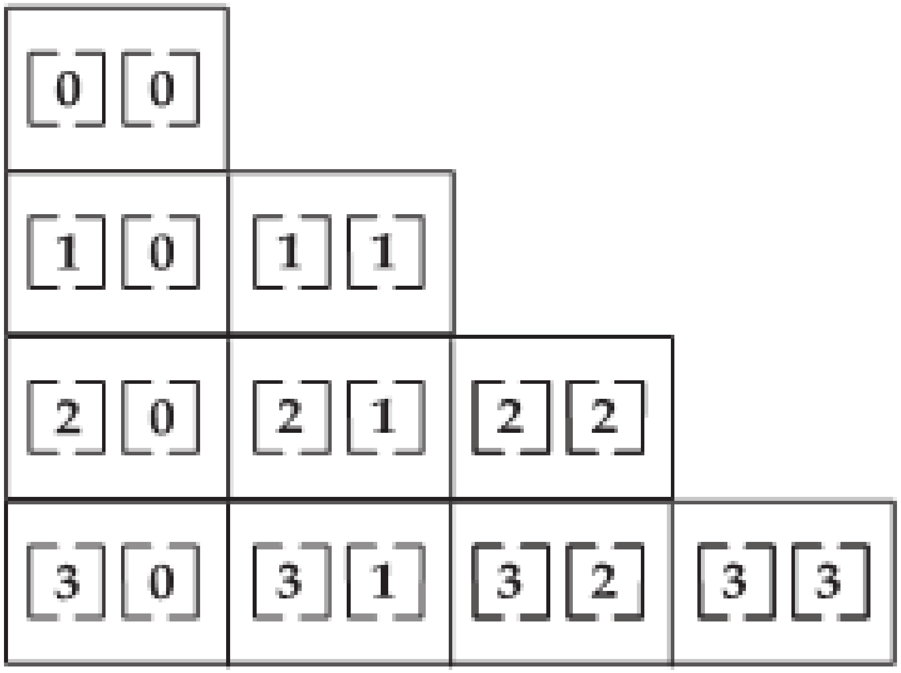
```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    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();
  }
}
```

# 2-D ARRAYS

This program generates the following output:

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

The array created by this program looks like this:



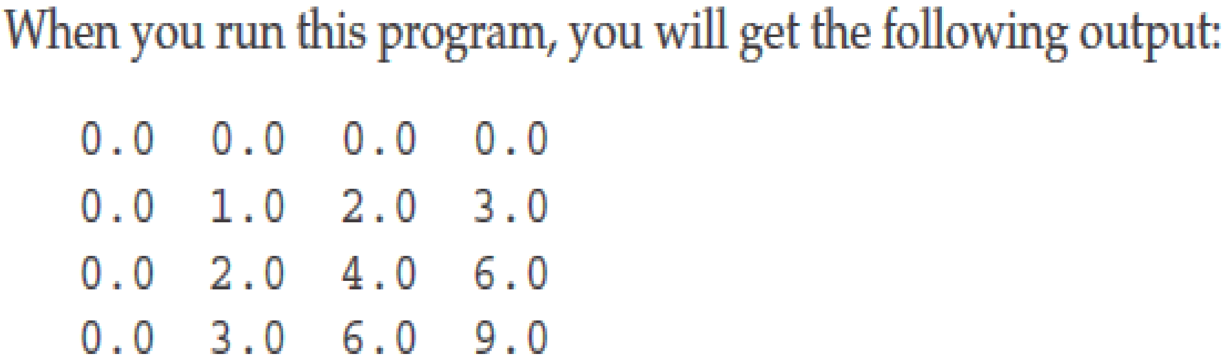
[0][0]<br>
[1][0] [1][1]<br>
[2][0] [2][1] [2][2]<br>
[3][0] [3][1] [3][2] [3][3]

# 2-D ARRAYS

```
// 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();
}
}
```

# 2-D ARRAYS

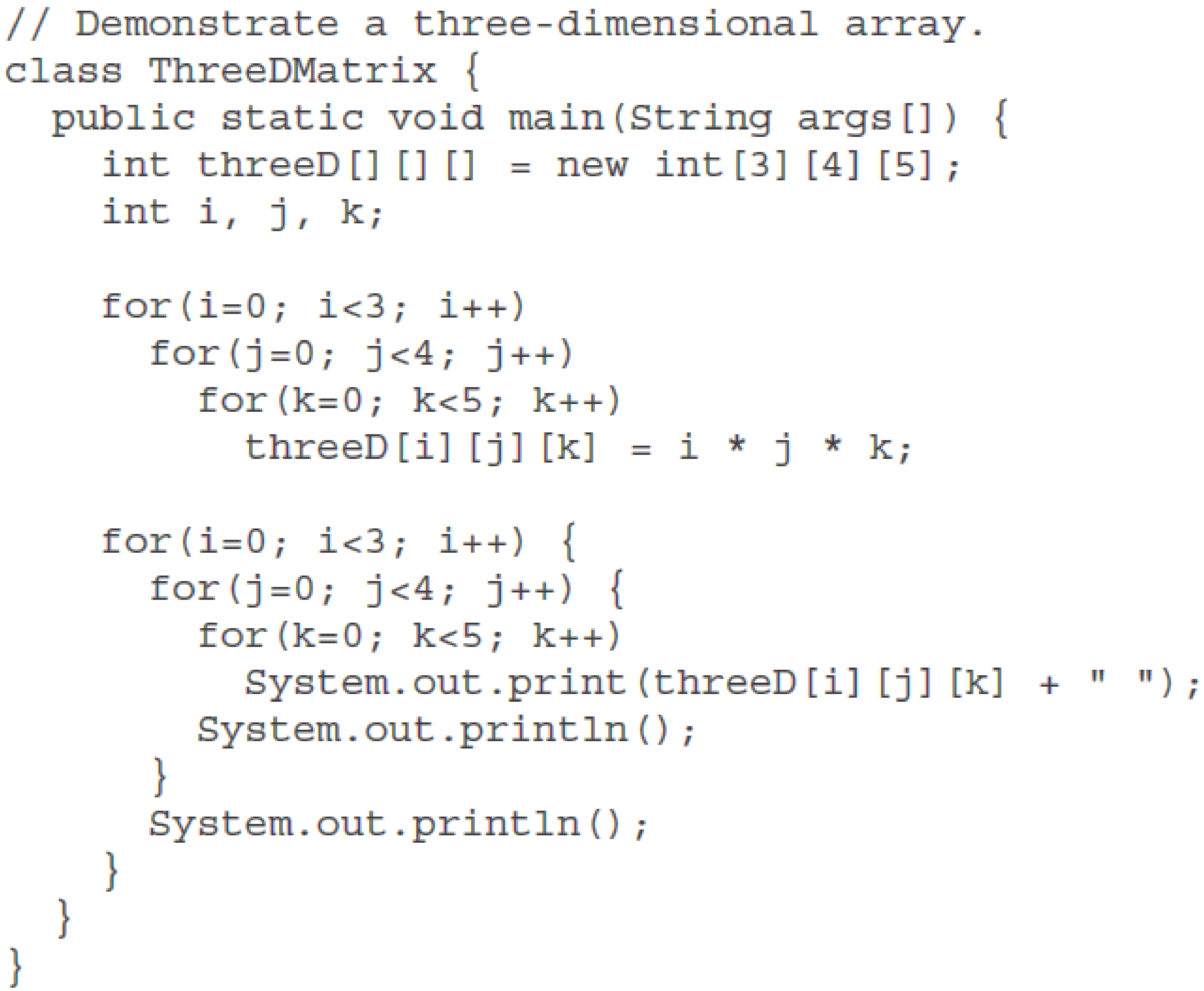
```
// Initialize a two-dimensional array.
class Matrix {
public static void main(String args[]) {
double m[][] = {
{ 0*0, 1*0, 2*0, 3*0 },
{ 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 },
{ 0*3, 1*3, 2*3, 3*3 }
};
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
System.out.print(m[i][j] + " ");
System.out.println();
}
}
```



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

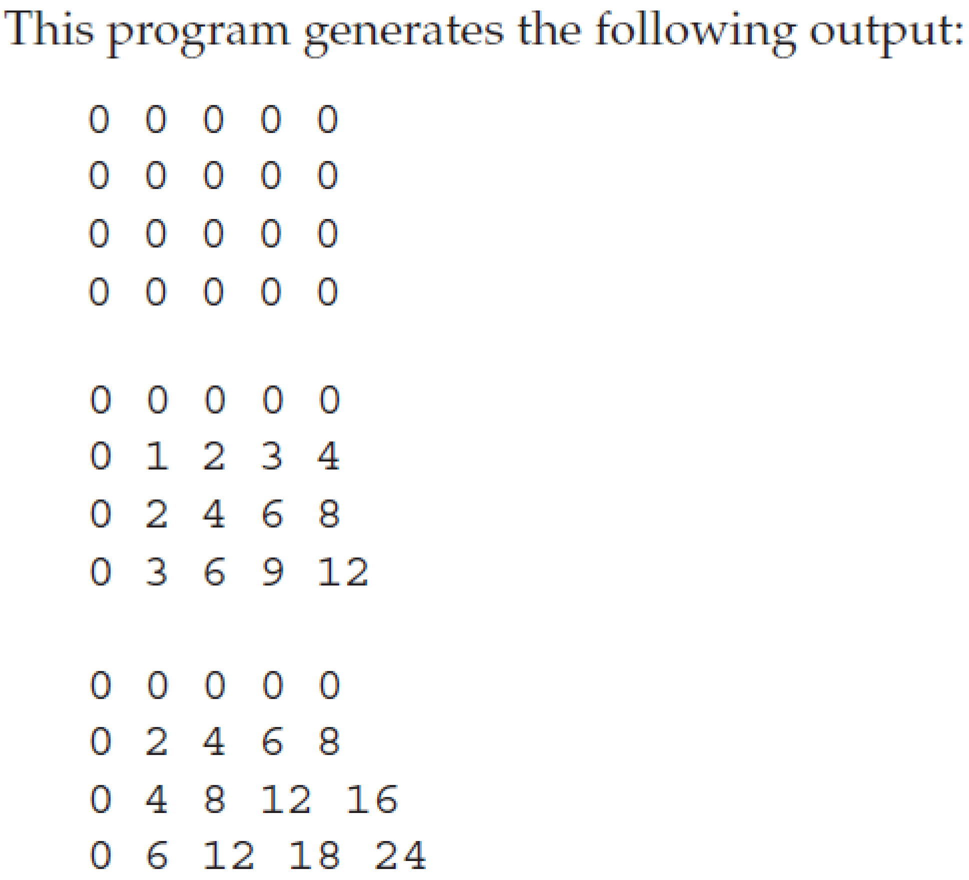
```
0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0
```

# 3-D ARRAYS



```
// Demonstrate a three-dimensional array.
class ThreeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
          threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

# 3-D ARRAYS

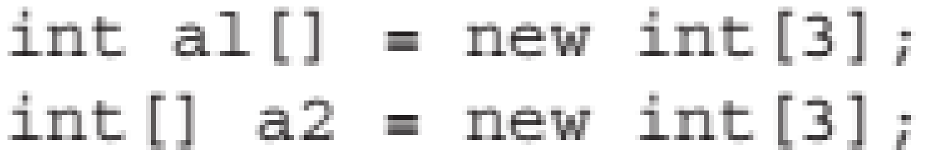


This program generates the following output:

```
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0

0  0  0  0  0
0  1  2  3  4
0  2  4  6  8
0  3  6  9  12

0  0  0  0  0
0  2  4  6  8
0  4  8  12  16
0  6  12  18  24
```

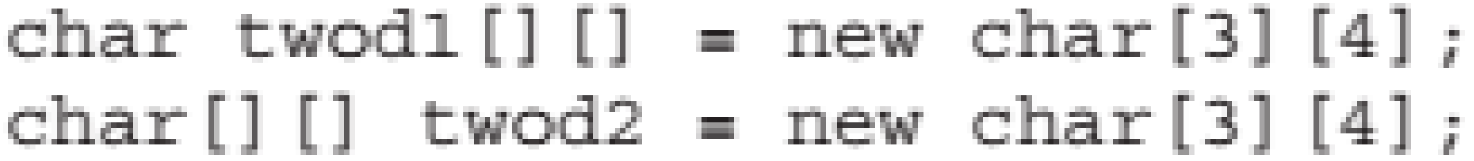
# ALTERNATIVE ARRAY DECLARATION

Following statements are equivalent



```
int a1[] = new int[3];
int[] a2 = new int[3];
```

Similarly



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

Declaration several arrays same time:

int[] nums1, nums2, nums3; // create three arrays

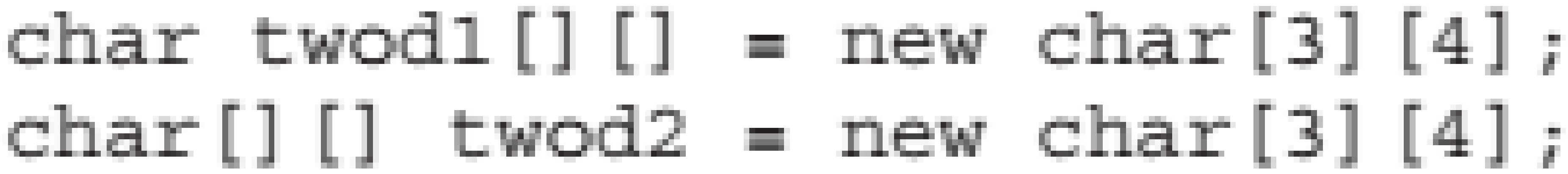
The alternative declaration form is also useful when specifying an array as a return type for a method.

# STRING

String is not a simple type, neither it is array of char

It defines an object

Similarly



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

Example:

String str = "this is a test";

System.out.println(str);

# A NOTE TO C/C++ PROGRAMMERS ABOUT POINTERS

Java does not support or allow pointers.

# DISCLAIMER

- These slides are not original and have been prepared from various sources for teaching purpose.

Sources:

- Herbert Schildt, Java$^{TM}$: The Complete Reference