Object Oriented Programming

Topics To Be Covered Today

- Array
  - Single & Multi-dimensional
- Java Operators
  - Assignment
  - Arithmetic
  - Relational
  - Logical
  - Bitwise & other

Arrays

- Arrays are:An array is a group of liked-typed variables referred to by a common name, with individual variables accessed by their index.

  1) declared
  2) created
  3) initialized
  4) used

- Also, arrays can have one or several dimensions.

Array Declaration

- Array declaration involves:
  1) declaring an array identifier
  2) declaring the number of dimensions
  3) declaring the data type of the array elements
- Two styles of array declaration:

  type array-variable[ ];

  or

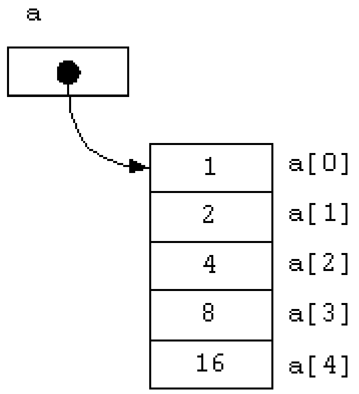
  type [ ] array-variable;

Array Creation

- After declaration, no array actually exists.
- In order to create an array, we use the **new** operator:

  type array-variable[ ];

  array-variable = new type[size];
- This creates a new array to hold size elements of type **type**, whose reference will be kept in the variable array-variable.

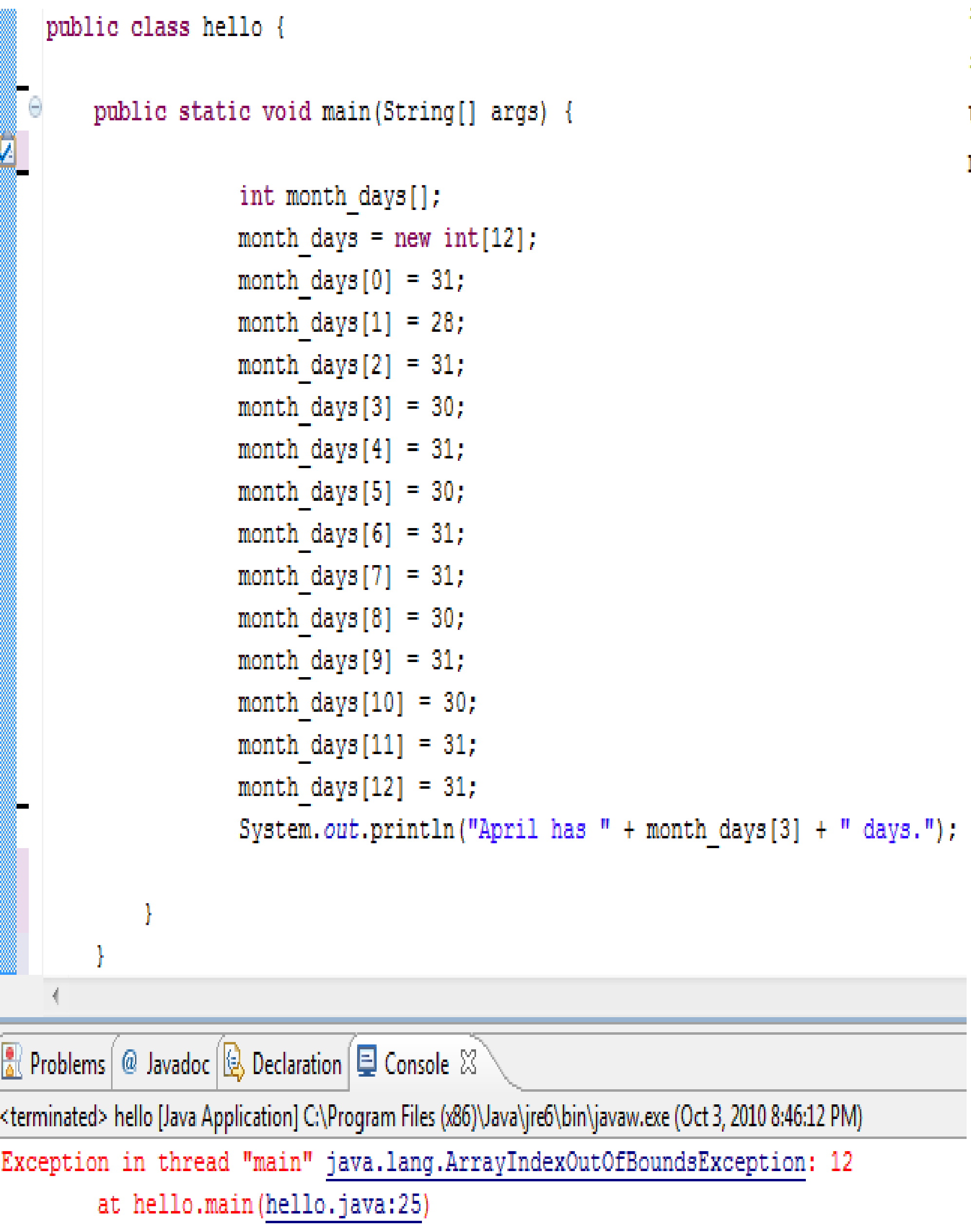


Array Indexing

- Later we can refer to the elements of this array through their indexes:

  array-variable[index]
- The array index always starts with zero!
- The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

JAVA

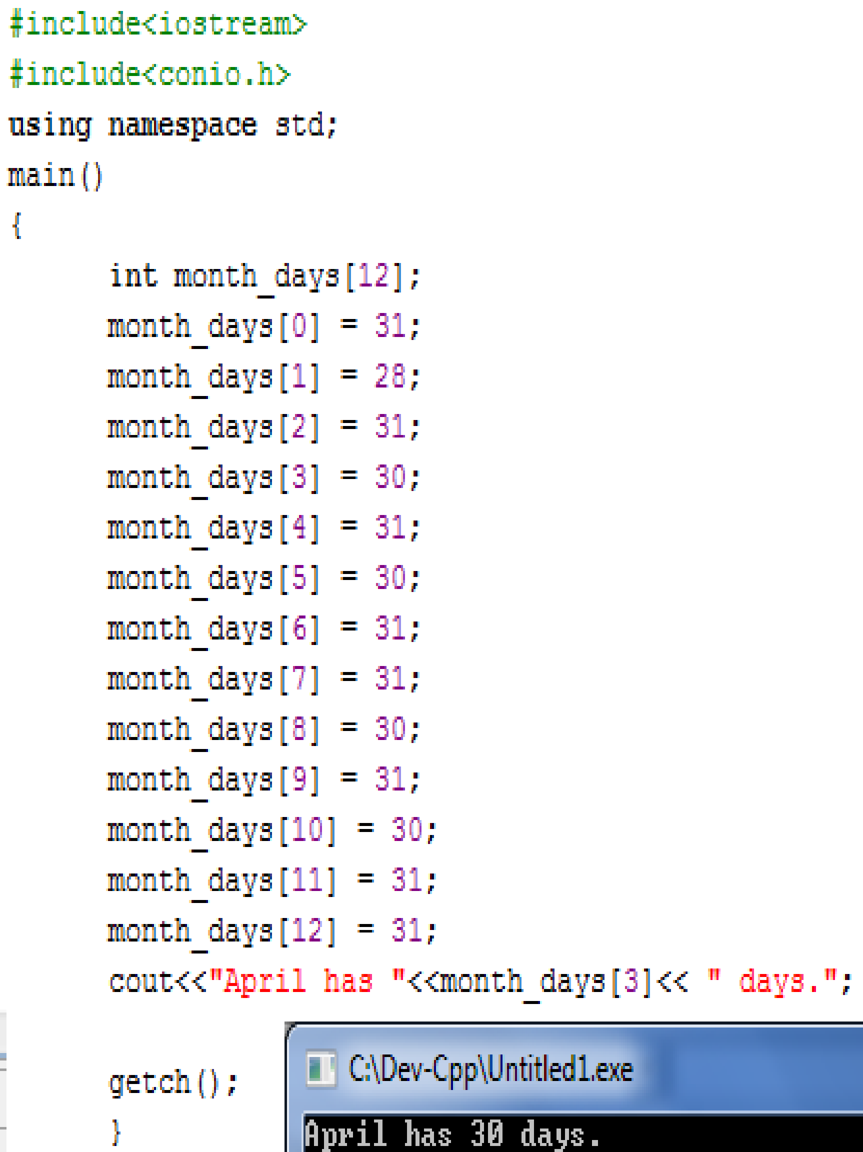


```
public class hello {

    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;
            month_days[12] = 31;
            System.out.println("April has " + month_days[3] + " days.");

    }
}
```

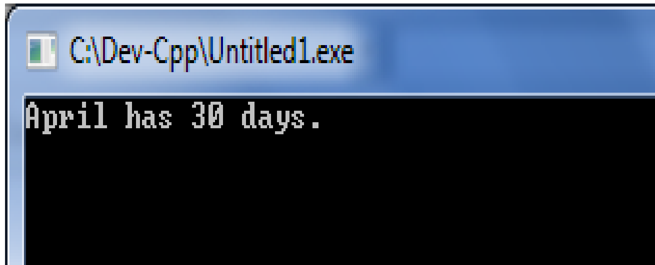
Problems | Javadoc | Declaration | Console

```
<terminated> hello [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Oct 3, 2010 8:46:12 PM)
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 12
	at hello.main(hello.java:25)
```

C++



```
#include<iostream>
#include<conio.h>
using namespace std;
main()
{
      int month_days[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;
      month_days[12] = 31;
      cout<<"April has "<<month_days[3]<< " days.";

      getch();
      }
```



C:\Dev-Cpp\Untitled1.exe

```
April has 30 days.
```

Example: Array Use

```
class Array {
      public static void main(String args[]) {
            int monthDays[];
            monthDays = new int[12];
            monthDays[0] = 31;
            monthDays[1] = 28;
            monthDays[2] = 31;
            monthDays[3] = 30;
            monthDays[4] = 31;
            monthDays[5] = 30;
            …
            monthDays[12] = 31;
            System.out.print(“April has ”);
            System.out.println(monthDays[3] +“ days.”);
      }
}
```

Array Initialization

- Arrays can be initialized when they are declared:

int monthDays[ ] = {31,28,31,30,31,30,31,31,30,31,30,31};

Comments:

1) there is no need to use the new operator
2) the array is created large enough to hold all specified elements

Example: Array Initialization

```
class Array {
     public static void main(String args[]) {
          int mthDys[]=
               {31,28,31,30,31,30,31,31,30,31,30,31};

          System.out.print("April ");
          System.out.println(mthDys[3]+ " days.");
     }
}
```

Multi-dimensional Array

- Multidimensional arrays are arrays of arrays:

  1) declaration

  ```
  int array[ ][ ];
  ```

  2) creation

  ```
  int array = new int[2][3];
  ```

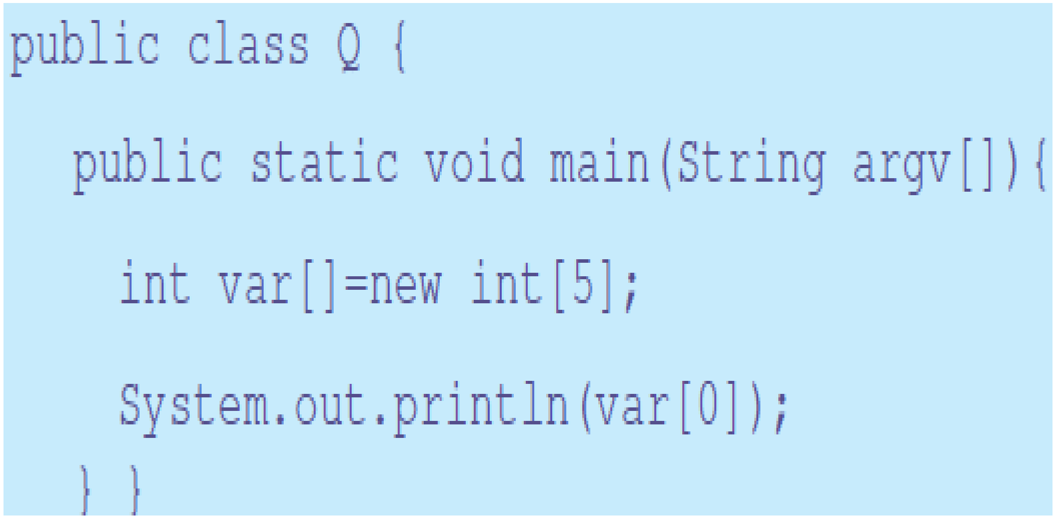
  3) initialization

  ```
  int array[ ][ ] = { {1, 2, 3}, {4, 5, 6} };
  ```

Example: Multi-dimensional Array

```
class Array {

  public static void main(String args[]) {

    int array[][] = { {1, 2, 3}, {4, 5, 6} };
    int i, j, k = 0;

    for(i=0; i<2; i++) {
          for(j=0; j<3; j++)
                System.out.print(array[i][j] + " ");
                System.out.println();
    }
  }
}
```

Class Participation



```
public class Q {
  public static void main(String argv[]){
    int var[]=new int[5];
    System.out.println(var[0]);
  } }
```

Java Operators

- Java operators are used to build value expressions.
- Java provides a rich set of operators:
  1) assignment
  2) arithmetic
  3) relational
  4) logical
  5) bitwise
  6) other

Operators and Operands

- Each operator takes one, two or three operands:

  1) a unary operator takes one operand

     j++;

  2) a binary operator takes two operands

     i = j++;

  3) a ternary operator requires three operands

     i = (i>12) ? 1 : i++;

Assignment Operator

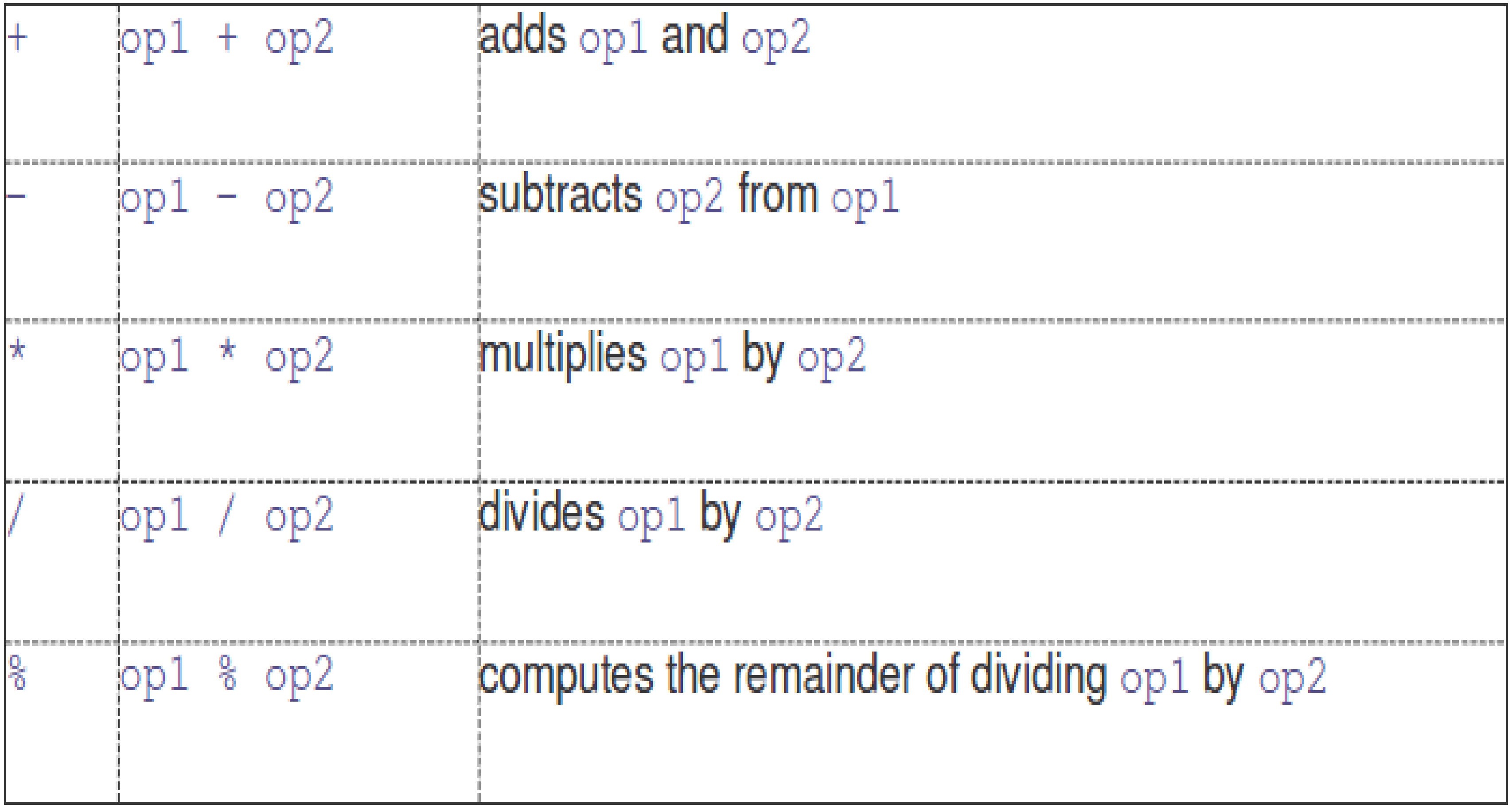
- A binary operator:

  variable = expression;
- It assigns the value of the expression to the variable.
- The types of the variable and expression must be compatible.
- The value of the whole assignment expression is the value of the expression on the right, so it is possible to chain assignment expressions as follows:
  - **int x, y, z;**
  - **x = y = z = 2;**

Arithmetic Operators

- Java supports various arithmetic operators for:
  1) integer numbers
  2) floating-point numbers
- There are two kinds of arithmetic operators:
  1) basic: addition, subtraction, multiplication, division and modulo
  2) shortcut: arithmetic assignment, increment and decrement

Basic Arithmetic Operator



+	op1 + op2	adds op1 and op2
-	op1 - op2	subtracts op2 from op1
*	op1 * op2	multiplies op1 by op2
/	op1 / op2	divides op1 by op2
%	op1 % op2	computes the remainder of dividing op1 by op2

Simple Arithmetic

```
public class Example {
  public static void main(String[] args) {
      int j, k, p, q, r, s, t;
      j = 5;
      k = 2;
      p = j + k;
      q = j - k;
      r = j * k;
      s = j / k;
      t = j % k;
      System.out.println("p = " + p);
      System.out.println("q = " + q);
      System.out.println("r = " + r);
      System.out.println("s = " + s);
      System.out.println("t = " + t);
  }
}
```

```
> java Example
p = 7
q = 3
r = 10
s = 2
t = 1
>
```

Arithmetic Assignment / Shorthand Operator

- Instead of writing

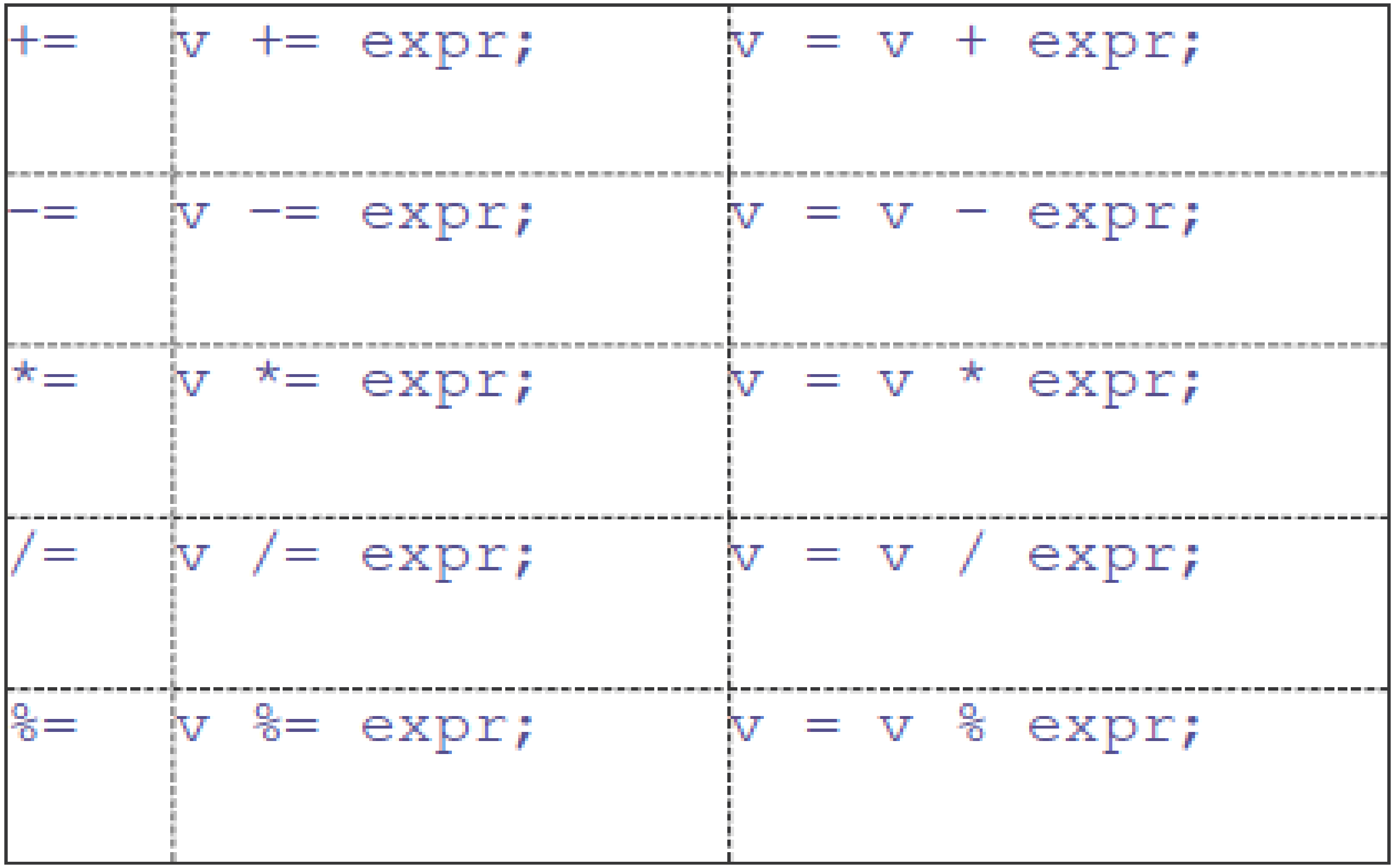
  variable = variable operator expression;
- for any arithmetic binary operator, it is possible to write shortly

  variable operator= expression;
- Benefits of the assignment operators:

  1) save some typing

  2) are implemented more efficiently by the Java run-time system

Arithmetic Assignment / Shorthand Operator



`+=`	`v += expr;`	`v = v + expr;`
`-=`	`v -= expr;`	`v = v - expr;`
`*=`	`v *= expr;`	`v = v * expr;`
`/=`	`v /= expr;`	`v = v / expr;`
`%=`	`v %= expr;`	`v = v % expr;`

Shorthand Operator

```
public class Example {
  public static void main(String[] args) {
      int j, p, q, r, s, t;
      j = 5;
      p = 1; q = 2; r = 3; s = 4; t = 5;
      p += j;
      q -= j;
      r *= j;
      s /= j;
      t %= j;
      System.out.println("p = " + p);
      System.out.println("q = " + q);
      System.out.println("r = " + r);
      System.out.println("s = " + s);
      System.out.println("t = " + t);
  }
}
```

```
> java Example
p = 6
q = -3
r = 15
s = 0
t = 0
>
```

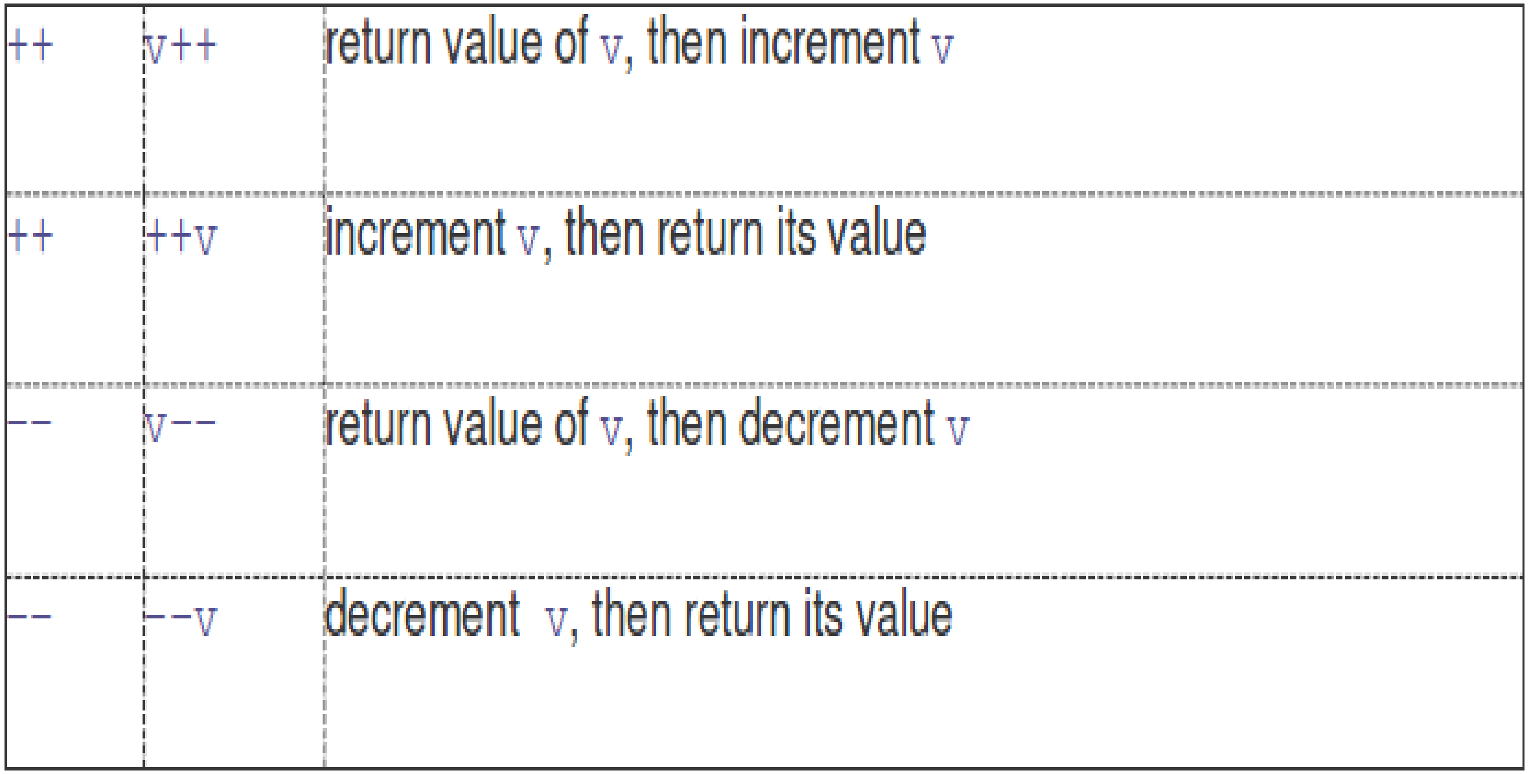
Increment/ Decrement Operators

- Two unary operators:

  1) ++ increments its operand by 1

  2) -- decrements its operand by 1
- The operand must be a numerical variable.
- Each operation can appear in two versions:
  - **prefix** version evaluates the value of the operand after performing the increment/decrement operation
  - **postfix** version evaluates the value of the operand before performing the increment/decrement operation

Increment/ Decrement



++	v++	return value of v, then increment v
++	++v	increment v, then return its value
--	v--	return value of v, then decrement v
--	--v	decrement v, then return its value

Increment and Decrement

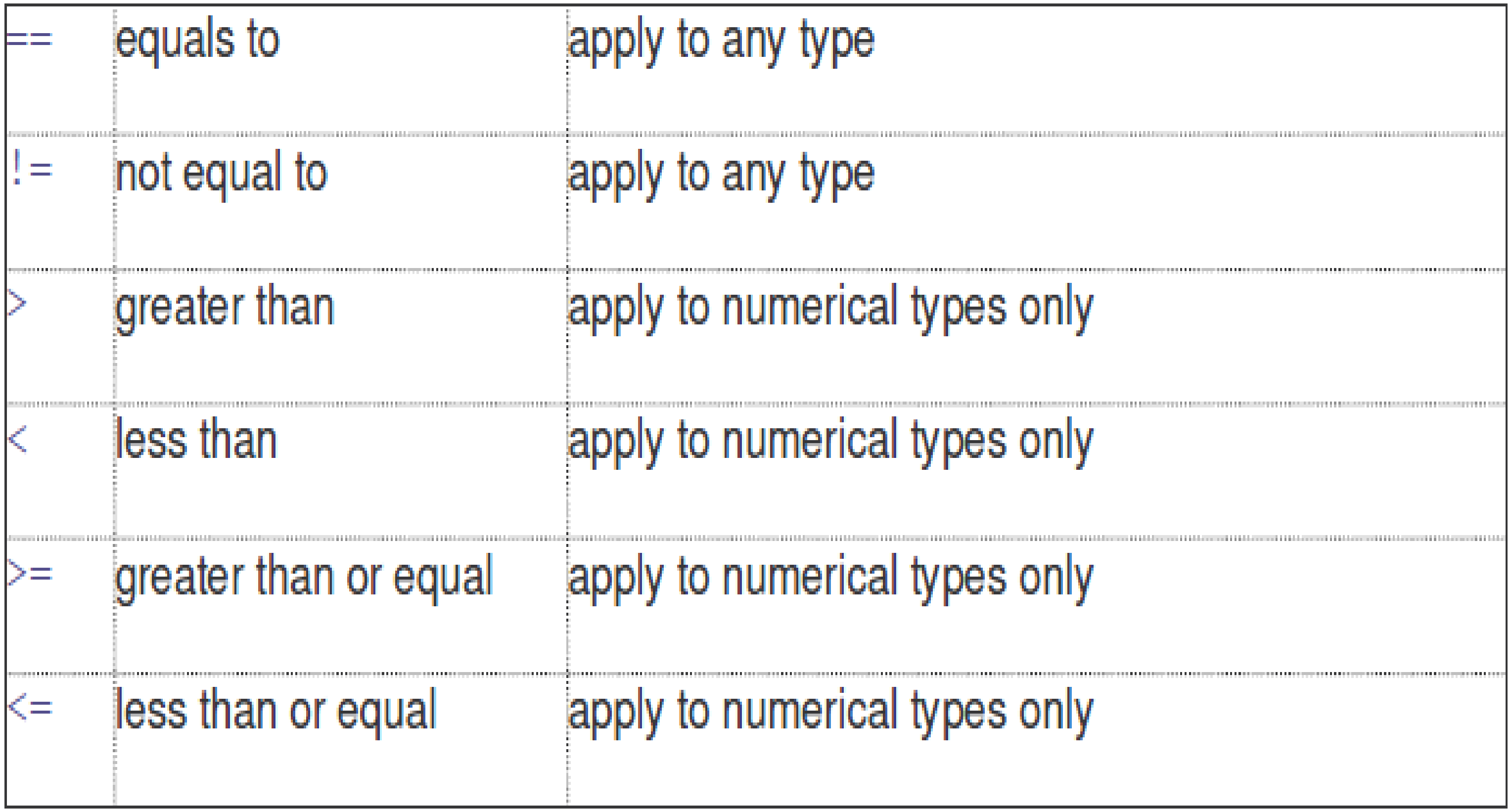
```
public class Example {
  public static void main(String[] args) {
    int j, p, q, r, s;
    j = 5;
    p = ++j;  //  j = j + 1;  p = j;
    System.out.println("p = " + p);
    q = j++;  //  q = j;      j = j + 1;
    System.out.println("q = " + q);
    System.out.println("j = " + j);
    r = --j;  //  j = j -1;   r = j;
    System.out.println("r = " + r);
    s = j--;  //  s = j;      j = j - 1;
    System.out.println("s = " + s);
  }
}
```

```
> java example
p = 6
q = 6
j = 7
r = 6
s = 6
>
```

Relational Operator

- Relational operators determine the relationship that one operand has to the other operand, specifically equality and ordering.
- The outcome is always a value of type **boolean**.
- They are most often used in branching and loop control statements.

Relational Operators



==	equals to	apply to any type
!=	not equal to	apply to any type
>	greater than	apply to numerical types only
<	less than	apply to numerical types only
>=	greater than or equal	apply to numerical types only
<=	less than or equal	apply to numerical types only

Relational Operator Examples

```
public class Example {
  public static void main(String[] args) {

       int p =2; int q = 2; int r = 3;


      System.out.println("p < r " + (p < r));
      System.out.println("p > r " + (p > r));
      System.out.println("p == q " + (p == q));
      System.out.println("p != q " + (p != q));

             }
}
```

```
> java Example
p < r true
p > r false
p == q true
p != q false

>
```

Logical Operators

- Logical operators act upon boolean operands only.
- The outcome is always a value of type boolean.
- In particular, 1and2 and 1or2 logical operators occur in two forms:

  1) full op1 & op2 and op1 | op2 where both op1 and op2 are evaluated

  2) short-circuit - op1 && op2 and op1 || op2 where op2 is only evaluated if the value of op1 is insufficient to determine the final outcome

Logical Operators

`&`	`op1 & op2`	logical `AND`
`|`	`op1 | op2`	logical `OR`
`&&`	`op1 && op2`	short-circuit `AND`
`||`	`op1 || op2`	short-circuit `OR`
`!`	`! op`	logical `NOT`
`^`	`op1 ^ op2`	logical `XOR`

Logical (&&) Operator Examples

```
public class Example {
  public static void main(String[] args) {
    boolean t = true;
    boolean f = false;

    System.out.println("f && f " + (f && f));
    System.out.println("f && t " + (f && t));
    System.out.println("t && f " + (t && f));
    System.out.println("t && t " + (t && t));

  }
}
```

```
> java Example
f && f false
f && t false
t && f false
t && t true
>
```

Logical (||) Operator Examples

```
public class Example {
  public static void main(String[] args) {
    boolean t = true;
    boolean f = false;

    System.out.println("f || f " + (f || f));
    System.out.println("f || t " + (f || t));
    System.out.println("t || f " + (t || f));
    System.out.println("t || t " + (t || t));

  }
}
```

```
> java Example
f || f false
f || t true
t || f true
t || t true
>
```

Logical (!) Operator Examples

```
public class Example {
  public static void main(String[] args) {
      boolean t = true;
      boolean f = false;

      System.out.println("!f " + !f);
      System.out.println("!t " + !t);

  }
}
```

```
> java Example
!f true
!t false
>
```

Logical Operator Examples

Short Circuiting with &&

```
public class Example {
  public static void main(String[] args) {
      boolean b;
      int j, k;

      j = 0; k = 0;
      b = ( j++ == k ) && ( j == ++k );
      System.out.println("b, j, k " + b + ", " + j + ", " + k);

      j = 0; k = 0;
      b = ( j++ != k ) && ( j == ++k );
      System.out.println("b, j, k " + b + ", " + j + ", " + k);
  }
}
```

```
> java Example
b, j, k true 1, 1
b, j, k false 1, 0
>
```

Logical Operator Examples

Short Circuiting with ||

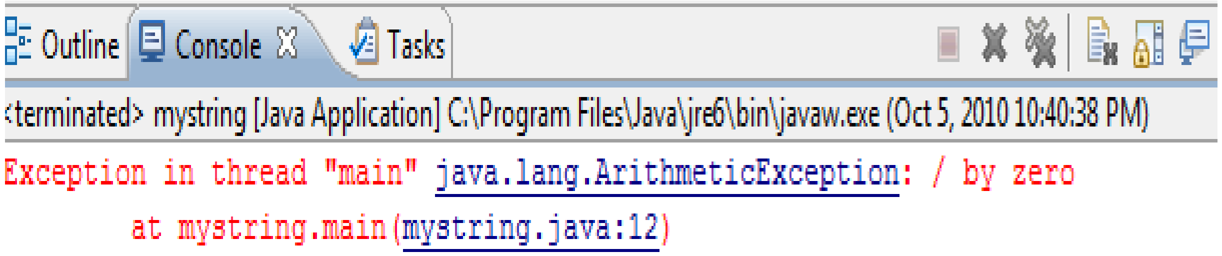
```
public class Example {
  public static void main(String[] args) {
      boolean b;
      int j, k;

      j = 0; k = 0;
      b = ( j++ == k ) || ( j == ++k );
      System.out.println("b, j, k " + b + ", " + j + ", " + k);

      j = 0; k = 0;
      b = ( j++ != k ) || ( j == ++k );
      System.out.println("b, j, k " + b + ", " + j + ", " + k);
  }
}
```

```
> java Example
b, j, k true 1, 0
b, j, k true 1, 1
>
```

Class Participation

```
class LogicalDemo {

  public static void main(String[] args) {
    int n = 2;
    if (n != 0 && n / 0 > 10)
         System.out.println("This is true");
    else
         System.out.println("This is false");
  }
```

Answer



```
<terminated> mystring [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Oct 5, 2010 10:40:38 PM)
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at mystring.main(mystring.java:12)
```

Bitwise Operators

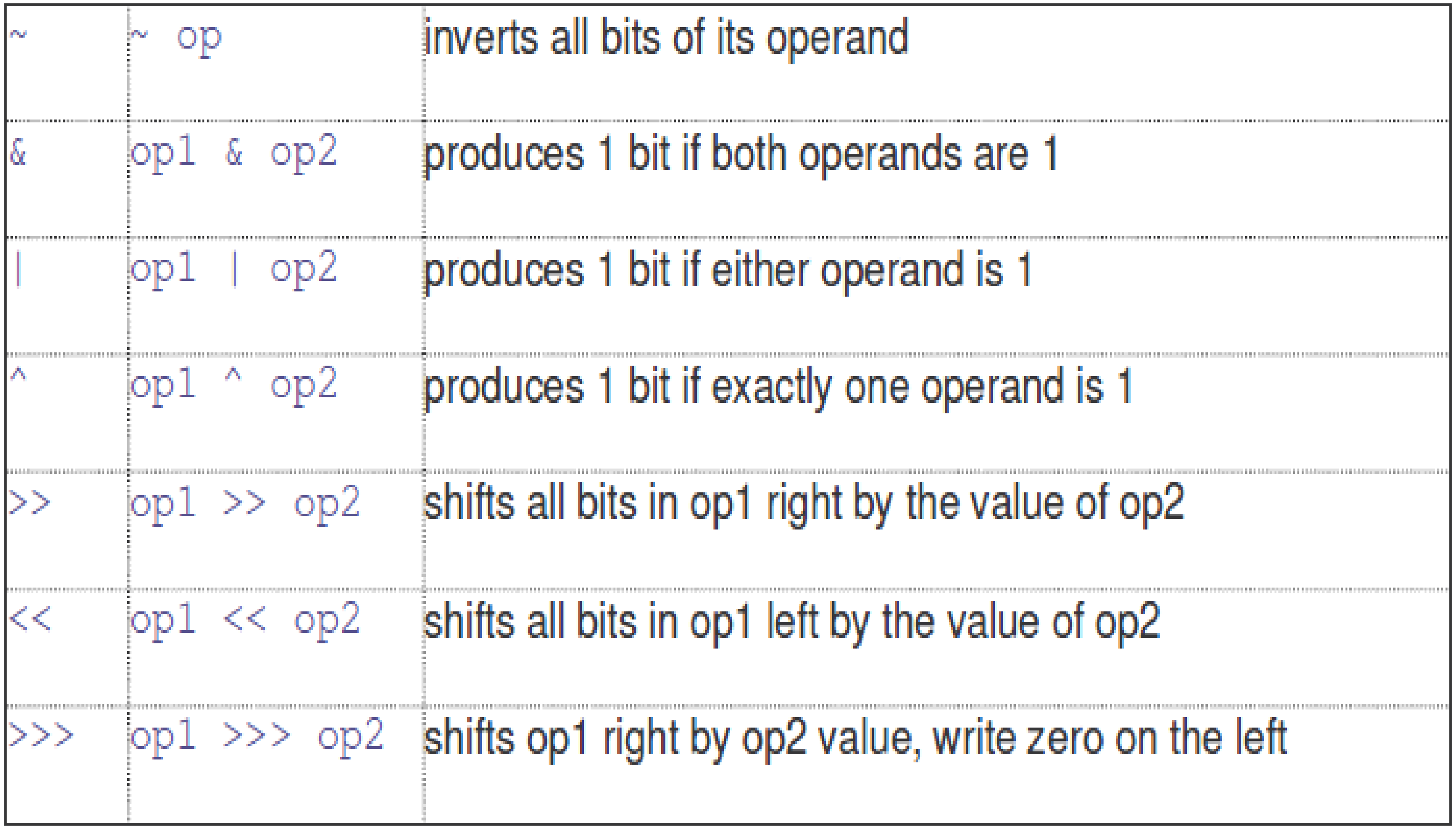
- Bitwise operators apply to integer types only.
- They act on individual bits of their operands.
- There are three kinds of bitwise operators:

1) basic bitwise AND, OR, NOT and XOR

2) shifts left, right and right-zero-fill

3) assignments bitwise assignment for all basic and shift operators

Bitwise Operators



`~`	`~ op`	inverts all bits of its operand
`&`	`op1 & op2`	produces 1 bit if both operands are 1
`|`	`op1 | op2`	produces 1 bit if either operand is 1
`^`	`op1 ^ op2`	produces 1 bit if exactly one operand is 1
`>>`	`op1 >> op2`	shifts all bits in op1 right by the value of op2
`<<`	`op1 << op2`	shifts all bits in op1 left by the value of op2
`>>>`	`op1 >>> op2`	shifts op1 right by op2 value, write zero on the left

Twos Complement Numbers

Base 10	A byte of binary
+127	01111111
+4	00000100
+3	00000011
+2	00000010
+1	00000001
+0	00000000
-1	11111111
-2	11111110
-3	11111101
-4	11111100
-128	10000000

Logical Operators (Bit Level)

& | ^ ~

```
int a = 10; // 00001010 = 10
int b = 12; // 00001100 = 12
```

&

AND

```
a       00000000000000000000000000001010    10
b       00000000000000000000000000001100    12
a & b   00000000000000000000000000001000     8
```

|

OR

```
a       00000000000000000000000000001010    10
b       00000000000000000000000000001100    12
a | b   00000000000000000000000000001110    14
```

^

XOR

```
a       00000000000000000000000000001010    10
b       00000000000000000000000000001100    12
a ^ b   00000000000000000000000000000110     6
```

~

NOT

```
 a      00000000000000000000000000001010    10
~a      11111111111111111111111111110101   -11
```

Logical (bit) Operator Examples

```
public class Example {
  public static void main(String[] args) {
    int a = 10;      // 00001010 = 10
    int b = 12;      // 00001100 = 12
    int and, or, xor, na;
    and = a & b;      // 00001000 = 8
    or  = a | b;      // 00001110 = 14
    xor = a ^ b;      // 00000110 = 6
    na  = ~a;         // 11110101 = -11
    System.out.println("and " + and);
    System.out.println("or " + or);
    System.out.println("xor " + xor);
    System.out.println("na " + na);
  }
}
```

```
> java Example
and 8
or 14
xor 6
na -11
>
```

Shift Operators (Bit Level)

<<   >>   >>>

• Shift Left	<<	Fill with Zeros
• Shift Right	>>	Based on Sign

Shift Operators << >>

```
int a =  3; // ...00000011 =  3
int b = -4; // ...11111100 = -4
```

<<

Left

```
a         00000000000000000000000000000011      3
a << 2    00000000000000000000000000001100     12

b         11111111111111111111111111111100     -4
b << 2    11111111111111111111111111110000    -16
```

>>

Right

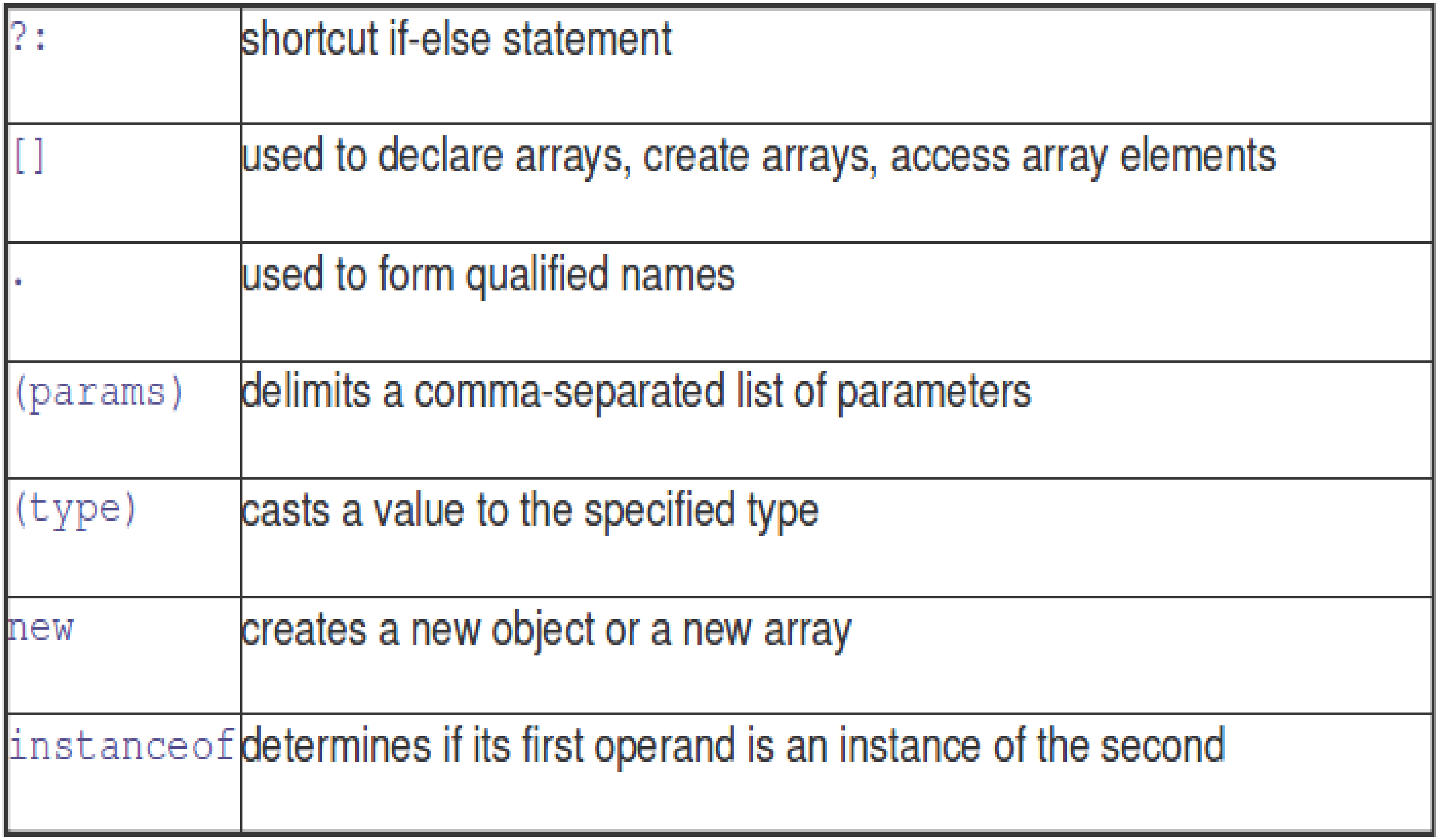
```
a         00000000000000000000000000000011      3
a >> 2    00000000000000000000000000000000      0

b         11111111111111111111111111111100     -4
b >> 2    11111111111111111111111111111111     -1
```

Shift Operator Examples

```
public class Example {
  public static void main(String[] args) {
    int a =  3;      // ...00000011 =  3
    int b = -4;      // ...11111100 = -4

    System.out.println("a<<2 = " + (a<<2));
    System.out.println("b<<2 = " + (b<<2));
    System.out.println("a>>2 = " + (a>>2));
    System.out.println("b>>2 = " + (b>>2));
    }
```

```
 > java Example
 a<<2 = 12
 b<<2 = -16
 a>>2 = 0
 b>>2 = -1
>
```

Other Operators



`?:`	shortcut if-else statement
`[]`	used to declare arrays, create arrays, access array elements
`.`	used to form qualified names
`(params)`	delimits a comma-separated list of parameters
`(type)`	casts a value to the specified type
`new`	creates a new object or a new array
`instanceof`	determines if its first operand is an instance of the second

Conditional Operators

- General form:

  expr1? expr2 : expr3

where:

1) expr1 is of type boolean
2) expr2 and expr3 are of the same type If expr1 is true, expr2 is evaluated, otherwise expr3 is evaluated.

Example: Conditional Operator

```
class Ternary {
  public static void main(String args[]) {
    int i, k;

    i = 10;
    k = i < 0 ? -i : i;
    System.out.print("Abs value of " + i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i;
    System.out.print("Abs value of " + i + " is " + k);
  }
}
```

Operator Precedence

- Java operators are assigned precedence order.
- Precedence determines that the expression

  1 + 2 * 6 / 3 > 4 && 1 < 0
- if equivalent to

  (((1 + ((2 * 6) / 3)) > 4) && (1 < 0))
- When operators have the same precedence, the earlier one binds stronger.

Operator Precedence

highest			
( )	[ ]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
lowest			

Questions?