Chapter 4
Operators

Operators

- Most of the operators divided in four groups:
  - Arithmetic
  - Bitwise
  - Relational
  - logical

Arithmetic Operators

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

Note: Arithmetic operators must be used with numeric and char data types.

Arithmetic Operators

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
public static void main(String args[]) {
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);
```

Arithmetic Operators

```
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}
```

Arithmetic Operators

- When you run this program, you will see the following output:

  Integer Arithmetic

  a = 2

  b = 6

  c = 1

  d = -1

  e = 1

  Floating Point Arithmetic

  da = 2.0

  db = 6.0

  dc = 1.5

  dd = -0.5

  de = 0.5

Arithmetic Operators – Modulus operator

- It can be applied to floating-point types as well as integer types

```
// Demonstrate the % operator.
class Modulus {
  public static void main(String args[]) {
    int x = 42;
    double y = 42.25;

    System.out.println("x mod 10 = " + x % 10);
    System.out.println("y mod 10 = " + y % 10);
  }
}
```

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

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

- Syntax:

var op = expression;

Example: a= a+4; → a+=4;

- Advantages:
  - they save you a bit of typing, because they are “shorthand”
  - implemented more efficiently by the Java run-time system

Arithmetic Compound Assignment Operators

```
// Demonstrate several assignment operators.
class OpEquals {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a += 5;
    b *= 4;
    c += a * b;
    c %= 6;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Arithmetic Operators – Increment & Decrement

- The increment operator increases its operand by one.
- Example: x=x+1; → x++;
- The decrement operator decreases its operand by one.
- Example: x=x-1; → x--;
- Used in *prefix* and *postfix* forms

Arithmetic Operators – Increment & Decrement

```
// Demonstrate ++.
class IncDec {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c;
    int d;
    c = ++b;
    d = a++;
    c++;
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
    System.out.println("d = " + d);
  }
}
```

The output of this program follows:

```
a = 2
b = 3
c = 4
d = 1
```

Bitwise Operators

- applied to the integer types, **long**, **int**, **short**, **char**, and **byte**

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise Operators

- Java uses an encoding known as *two's complement*, which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result.
- For example,
- –42 is represented by inverting all of the bits in 42, or 00101010, which yields 11010101, then adding 1, which results in 11010110, or –42.
- To decode a negative number, first invert all of the bits, then add 1. For example, –42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

Bitwise Operators

The Bitwise Logical Operators

The bitwise logical operators are &, |, ^, and ~. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the *bitwise complement*, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

00101010

becomes

11010101

after the NOT operator is applied.

Bitwise Operators

The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
  00101010      42
& 00001111      15
----------
  00001010      10
```

Bitwise Operators

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
  00101010      42
| 00001111      15
----------
  00101111      47
```

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

```
  00101010      42
^ 00001111      15
----------
  00100101      37
```

```
// Demonstrate the bitwise logical operators.
class BitLogic {
  public static void main(String args[]) {
    String binary[] = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };
    int a = 3; // 0 + 2 + 1 or 0011 in binary
    int b = 6; // 4 + 2 + 0 or 0110 in binary
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b) | (a & ~b);
    int g = ~a & 0x0f;

    System.out.println("        a = " + binary[a]);
    System.out.println("        b = " + binary[b]);
    System.out.println("      a|b = " + binary[c]);
    System.out.println("      a&b = " + binary[d]);
    System.out.println("      a^b = " + binary[e]);
    System.out.println("~a&b|a&~b = " + binary[f]);
    System.out.println("       ~a = " + binary[g]);
  }
}
```

Bitwise Operators

```
                a  =  0011
                b  =  0110
              a|b  =  0111
              a&b  =  0010
              a^b  =  0101
~a&b | a&~b  =  0101
               ~a  =  1100
```

Bitwise Operators- Left Shift

- left shift operator, **<<**, shifts all of the bits in a value to the left a specified number of times.
- It has this general form:

  value << *num*

- For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.
- 23<<1

```
    00010111 (decimal +23) LEFT-SHIFT
=   00101110 (decimal +46)
```

Bitwise Operators- Left Shift

```
// Left shifting a byte value.
class ByteShift {
public static void main(String args[]) {
    byte a = 64, b;
    int i;
    i = a << 2;
    b = (byte) (a << 2);
    System.out.println("Original value of a: " + a);
    System.out.println("i and b: " + i + " " + b);
}
}
```

- The output generated by this program is shown here:

```
Original value of a: 64
i and b: 256 0
```

Bitwise Operators- Left Shift

```
// Left shifting as a quick way to multiply by 2.
class MultByTwo {
  public static void main(String args[]) {
    int i;
    int num = 0xFFFFFFE;

    for(i=0; i<4; i++) {
      num = num << 1;
      System.out.println(num);
    }
  }
}
```

The program generates the following output:

```
536870908
1073741816
2147483632
-32
```

Bitwise Operators- Right Shift

- The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times.
- Its general form is shown here:

  value >> *num*

- Example,

  int a = 32;

  a = a >> 2; // a now contains 8

- When a value has bits that are "shifted off," those bits are lost.

00100011	35
>> 2	
00001000	8

11111000	–8
>>1	
11111100	–4

Bitwise Operators- Unsigned Right Shift

- right operator, **>>>**, which always shifts zeros into the high-order bit

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

11111111 11111111 11111111 11111111 –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int

Bitwise Operators- Compound Assignments

- a = a >> 4; → a >>= 4;
- a = a | b; → a |= b;

```
class OpBitEquals {
public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;
        a |= 4;
        b >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
}
}
```

The output of this program is shown here:

```
        a = 3
        b = 1
        c = 6
```

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Relational Operators

As stated, the result produced by a relational operator is a **boolean** value. For example, the following code fragment is perfectly valid:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

In this case, the result of a<b (which is false) is stored in c.

Relational Operators

```
int done;
// ...
if(!done) ... // Valid in C/C++
if(done) ...   // but not in Java.
```

In Java, these statements must be written like this:

```
if(done == 0) ... // This is Java-style.
if(done != 0) ...
```

The reason is that Java does not define true and false in the same way as C/C++. In C/C++, true is any nonzero value and false is zero. In Java, true and false are nonnumeric values that do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

Boolean Logical Operators

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

Boolean Logical Operators

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Boolean Logical Operators

```
// Demonstrate the boolean logical operators.
class BoolLogic {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;
    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
    boolean g = !a;
    System.out.println("        a = " + a);
    System.out.println("        b = " + b);
    System.out.println("      a|b = " + c);
    System.out.println("      a&b = " + d);
    System.out.println("      a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println("       !a = " + g);
  }
}
```

```
        a = true
        b = false
      a|b = true
      a&b = false
      a^b = true
a&b|a&!b = true
       !a = false
```

Boolean Logical Operators- Short-Circuit Logical Operators

- Short circuit AND - &&
- Short circuit OR - ||

- Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.
- Example:

```
if (denom != 0 && num / denom > 10)
```

- if(c==1 && e++ < 100) d = 100;

Assignment Operator

- *Syntax:*
  - *var = expression;*
- Example:
- int x, y, z;
- x = y = z = 100; // set x, y, and z to 100

? Operator

- ***ternary* (three-way) *operator*** that can replace certain types of if-then-else statements
- The **?** has this general form:

  expression1 **?** *expression2* **:** *expression3*

- Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated.
- Both *expression2* and *expression3* are required **to return the same type**, which can't be **void.**
- Example:

```
ratio = denom == 0 ? 0 : num / denom;
```

? Operator

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

    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
  }
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operator Precedence

TABLE 4-1
The Precedence of the Java Operators

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

disclaimer

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

Sources:

- Herbert Schildt, Java™: The Complete Reference

Thank You….