IT602: Object-Oriented Programming

Lecture - 05

Operators & Expressions and Control Flow

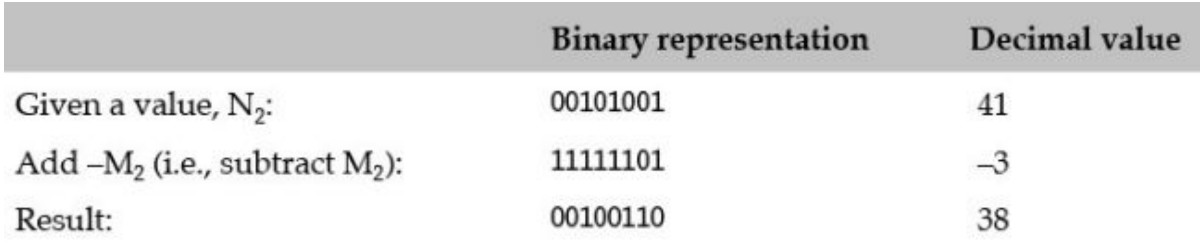
Arpit Rana

1st Feb 2022

Representing Integers

Integer data type in Java represents signed integer values (i.e. both +ve and -ve integer values)

- A value of type ***byte*** requires 8 bits to represent 256 values.
- Java uses *two’s complement* to store signed values of integer data types, i.e. ***-128 (-2^7)*** to ***+ 127 (2^7-1)*** inclusive.

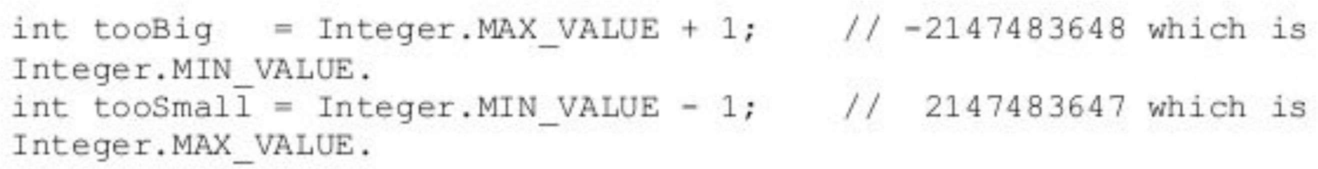


	Binary representation	Decimal value
Given a value, N_2:	00101001	41
Add $-M_2$ (i.e., subtract M_2):	11111101	–3
Result:	00100110	38

- This equally applies to values of other integer types: ***short, int, long*** with 16, 32, and 64 bits respectively.

Arithmetic Operators: *, /, %, +, -

- Integer values wrap around and no overflow and underflow is indicated



```
int tooBig   = Integer.MAX_VALUE + 1;    // -2147483648 which is
Integer.MIN_VALUE.
int tooSmall = Integer.MIN_VALUE - 1;    //  2147483647 which is
Integer.MAX_VALUE.
```

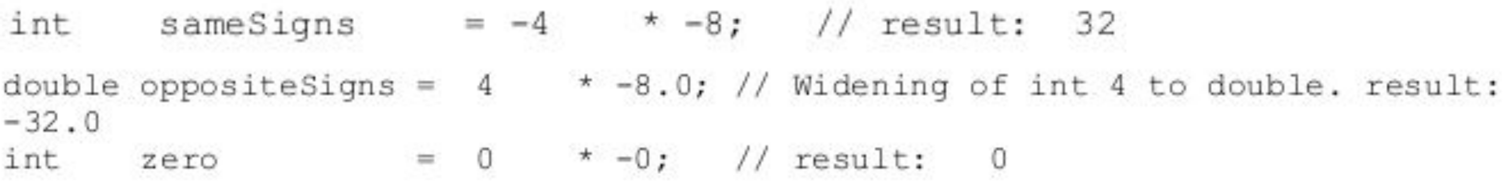
- What about the equivalent expression in ***`byte`*** type?

Arithmetic Operators: *, /, %, +, -

The unary operators have the highest precedence of all arithmetic operators.

```
int value = - -10;                 // (-(-10)) is 10
```

Multiplication operator *

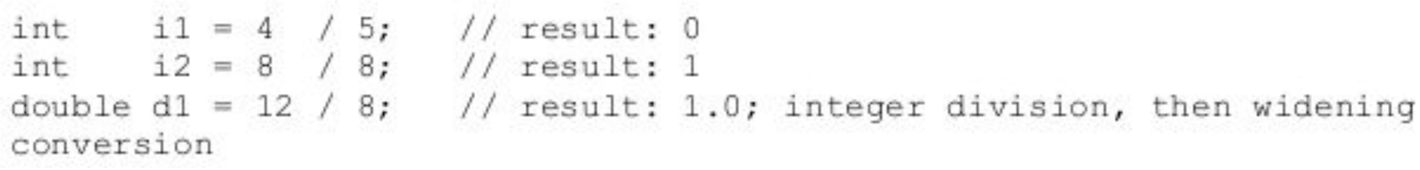


```
int     sameSigns      = -4     * -8;    // result:  32
double oppositeSigns =  4     * -8.0; // Widening of int 4 to double. result:
-32.0
int     zero           =  0     * -0;    // result:   0
```

Arithmetic Operators: *, /, %, +, -

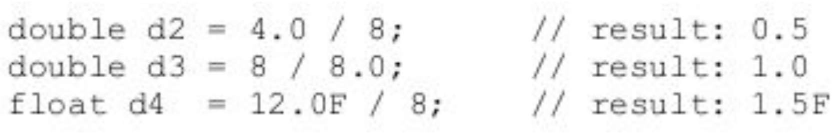
Division operator /

- If operands are integral, the operation results in integer division.



```
int    i1 = 4  / 5;   // result: 0
int    i2 = 8  / 8;   // result: 1
double d1 = 12 / 8;   // result: 1.0; integer division, then widening
conversion
```

- If any of the operands is a floating-point type, the operation performs floating-point division.



```
double d2 = 4.0 / 8;       // result: 0.5
double d3 = 8 / 8.0;       // result: 1.0
float d4  = 12.0F / 8;     // result: 1.5F
```

Arithmetic Operators: *, /, %, +, -

Remainder operator %

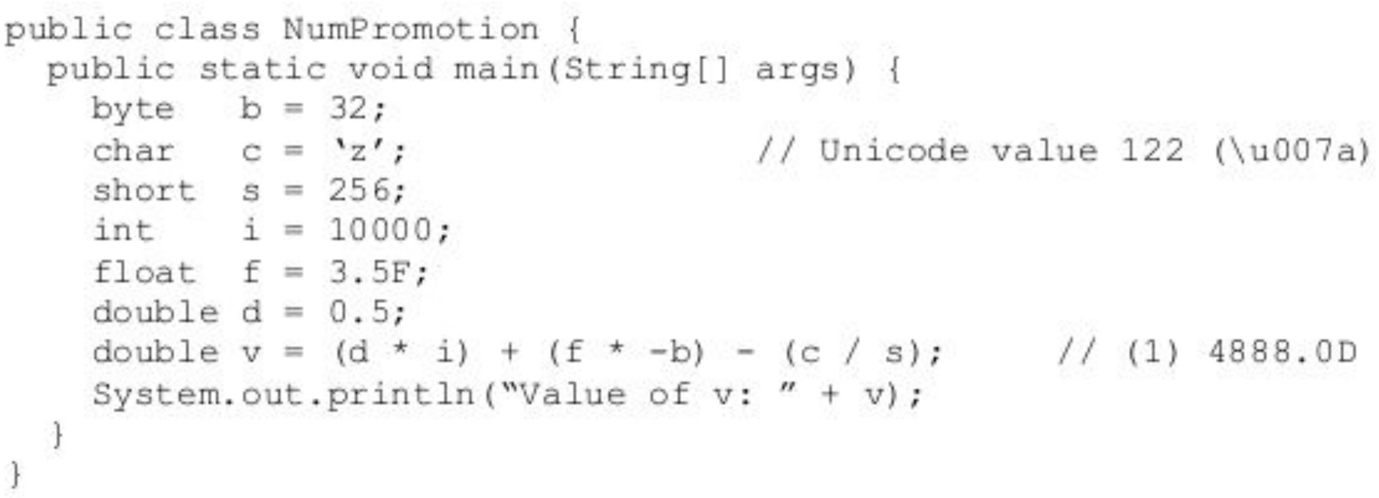
- The remainder can be negative only if the dividend is negative, and the sign of the divisor is irrelevant.
    - 7 % 5 = 2
    - 7 % -5 = 2
    - -7 % 5 = -2
    - -7 % -5 = -2

- The remainder operator also accepts the operands is a floating-point type.

Numeric Promotions in Arithmetic Expressions

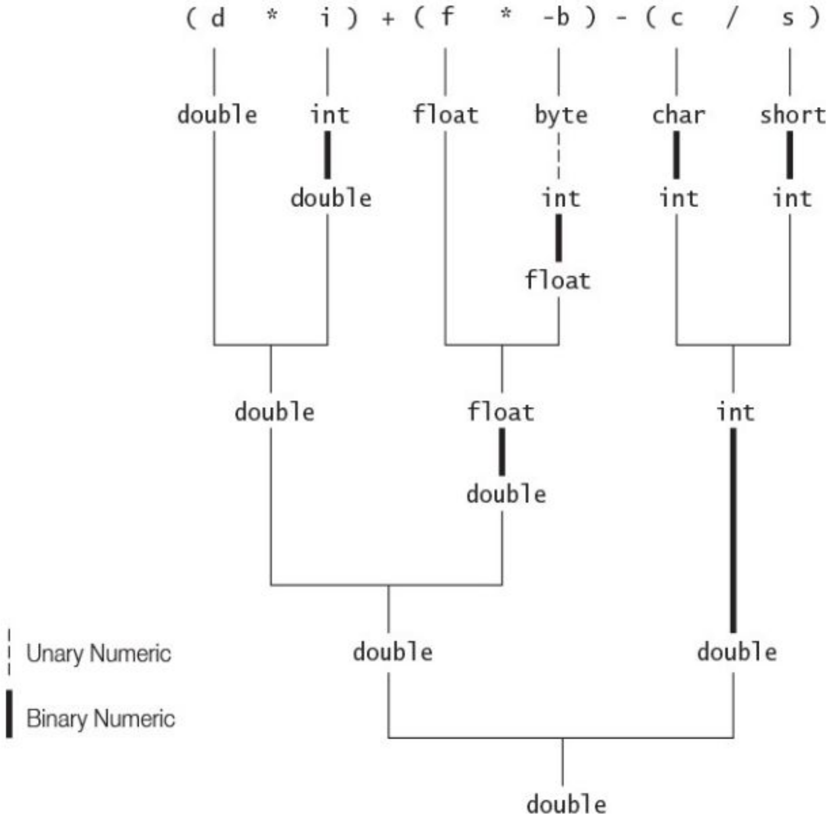
Numeric Promotions

- This is applied to operands of binary arithmetic operators which leads to type promotion for the operands.
- The result is of the promoted type, which is always type ***`int`*** or wider.

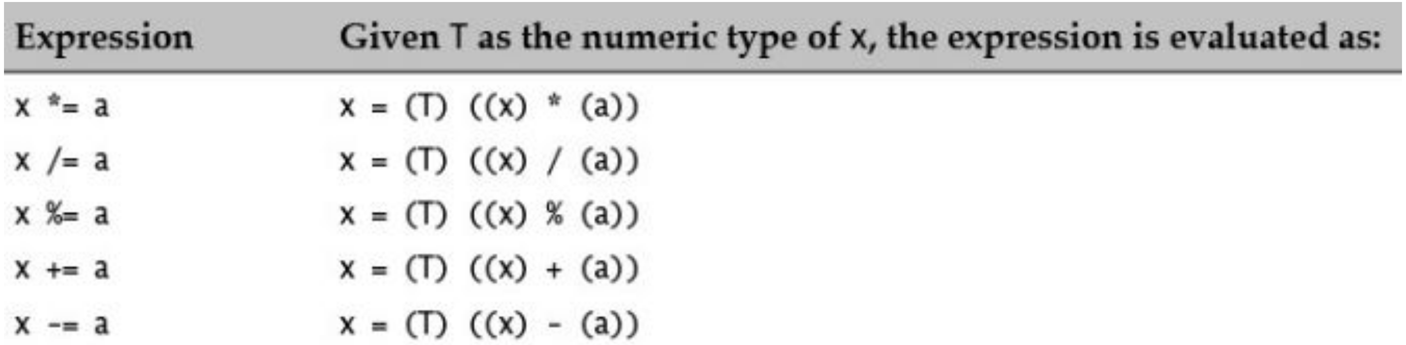


```
public class NumPromotion {
  public static void main(String[] args) {
    byte   b = 32;
    char   c = ‘z’;                          // Unicode value 122 (\u007a)
    short  s = 256;
    int    i = 10000;
    float  f = 3.5F;
    double d = 0.5;
    double v = (d * i) + (f * -b) - (c / s);      // (1) 4888.0D
    System.out.println(“Value of v: ” + v);
  }
}
```

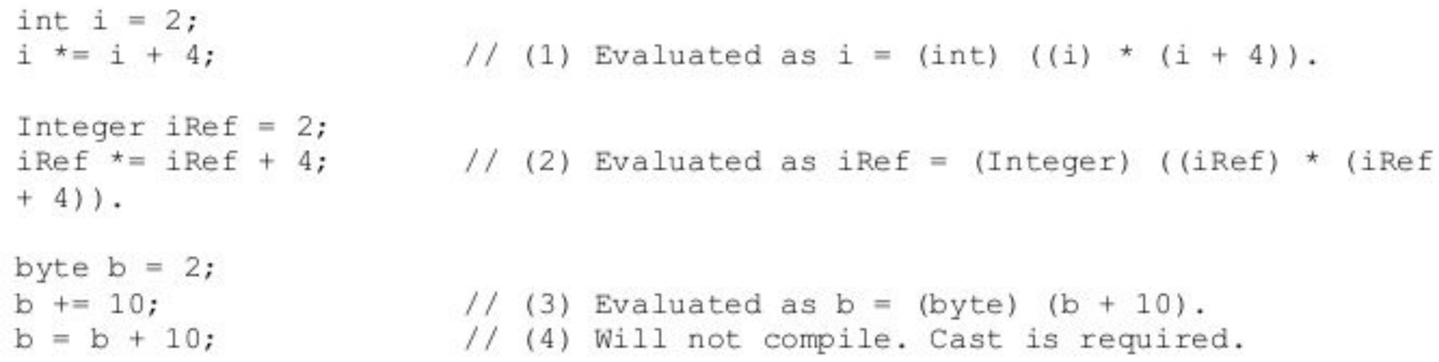
Numeric Promotions in Arithmetic Expressions



Arithmetic Compound Assignment Operators



Expression	Given T as the numeric type of x, the expression is evaluated as:
x *= a	x = (T) ((x) * (a))
x /= a	x = (T) ((x) / (a))
x %= a	x = (T) ((x) % (a))
x += a	x = (T) ((x) + (a))
x -= a	x = (T) ((x) - (a))

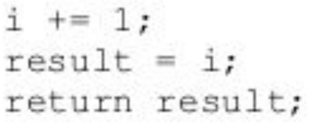


```
int i = 2;
i *= i + 4;             // (1) Evaluated as i = (int) ((i) * (i + 4)).

Integer iRef = 2;
iRef *= iRef + 4;       // (2) Evaluated as iRef = (Integer) ((iRef) * (iRef
+ 4)).

byte b = 2;
b += 10;                // (3) Evaluated as b = (byte) (b + 10).
b = b + 10;             // (4) Will not compile. Cast is required.
```

Variable Increment and Decrement Operators

Prefix operator

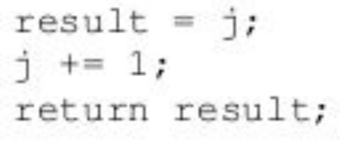
- The prefix increment operator: ***++i*** adds 1 to the value of ***i***, stores the new value in ***i***, and returns the new value as the value of the expression.



```
i += 1;
result = i;
return result;
```

Postfix operator

- The postfix increment operator: ***j++*** adds 1 to the value of ***j***, stores the new value in ***j***, and returns the original value that was in ***j*** as the value of the expression.

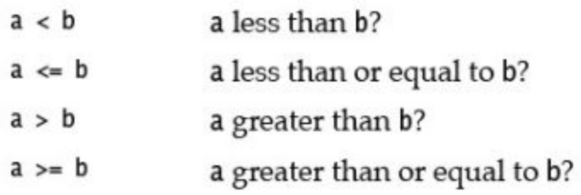


```
result = j;
j += 1;
return result;
```

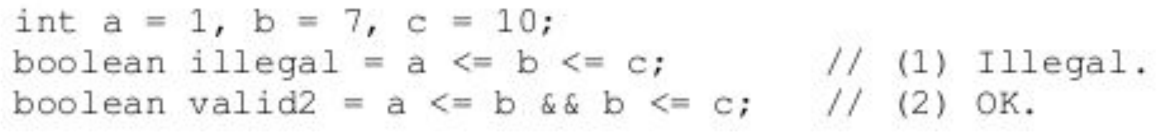
Boolean Expressions

Relational operators

- Given that ***a*** and ***b*** represent numeric expressions, the relational (also called comparison) operators are defined as below.



`a < b`	a less than b?
`a <= b`	a less than or equal to b?
`a > b`	a greater than b?
`a >= b`	a greater than or equal to b?



```
int a = 1, b = 7, c = 10;
boolean illegal = a <= b <= c;          // (1) Illegal.
boolean valid2 = a <= b && b <= c;      // (2) OK.
```

Boolean Expressions

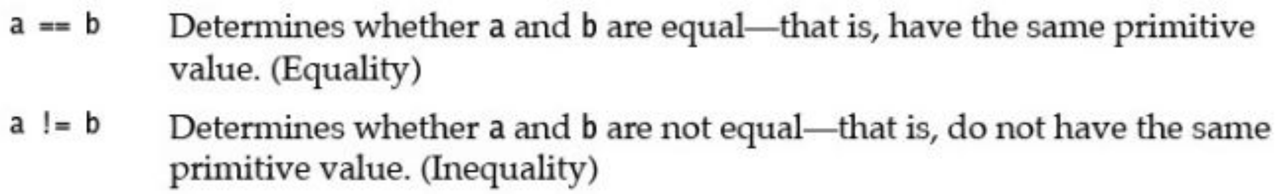
Equality

- The equality operators have lower precedence than the relational operators, but higher precedence than the assignment operators.
- We distinguish between *primitive data equality, object reference equality, and object value equality*.

Boolean Expressions

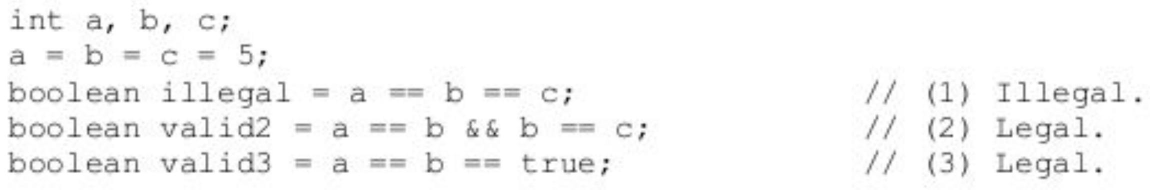
Primitive Data Value Equality: ==, !=

- Given that *a* and *b* represent operands of primitive data types, the primitive data value equality operators are defined as below.



`a == b` Determines whether `a` and `b` are equal—that is, have the same primitive value. (Equality)

`a != b` Determines whether `a` and `b` are not equal—that is, do not have the same primitive value. (Inequality)

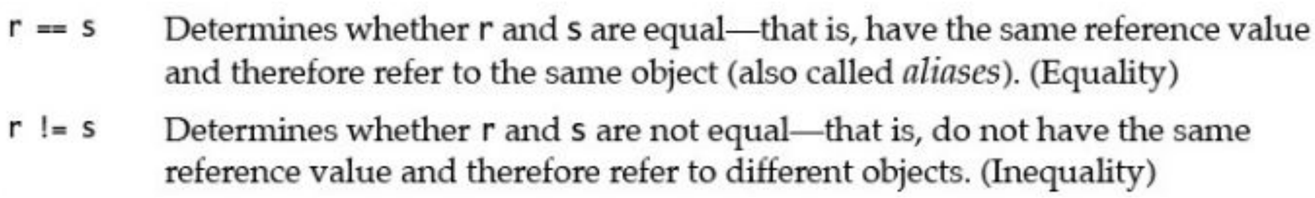


```
int a, b, c;
a = b = c = 5;
boolean illegal = a == b == c;                // (1) Illegal.
boolean valid2 = a == b && b == c;            // (2) Legal.
boolean valid3 = a == b == true;              // (3) Legal.
```

Boolean Expressions

Object Reference Equality: ==, !=

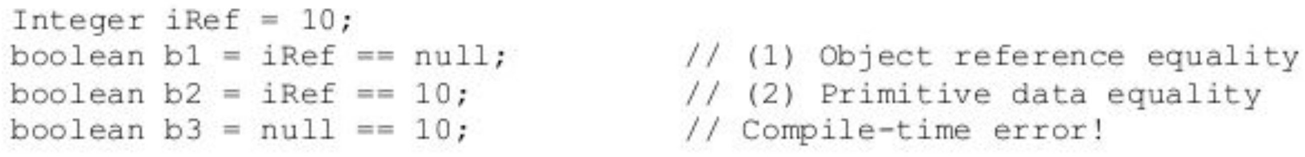
- Given that *a* and *b* are reference variables, the reference equality operators are defined as below.



`r == s` Determines whether `r` and `s` are equal—that is, have the same reference value and therefore refer to the same object (also called *aliases*). (Equality)

`r != s` Determines whether `r` and `s` are not equal—that is, do not have the same reference value and therefore refer to different objects. (Inequality)

- When the type of both the operands is either a reference type of the null type, operators test for reference equality; otherwise, they test for primitive data equality.

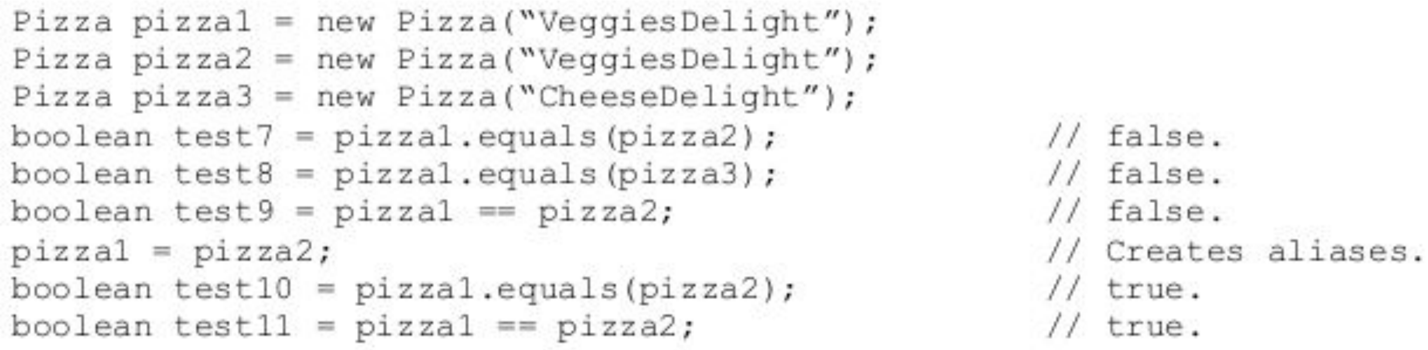


```
Integer iRef = 10;
boolean b1 = iRef == null;              // (1) Object reference equality
boolean b2 = iRef == 10;                // (2) Primitive data equality
boolean b3 = null == 10;                // Compile-time error!
```

Boolean Expressions

Object Value Equality

- The Object class provides the method ***public boolean equals(Object obj)***.
- `java.lang.String` and the wrapper classes for the primitive data types override this method for *deep comparison* (e.g. whether strings contain identical character sequences) check.

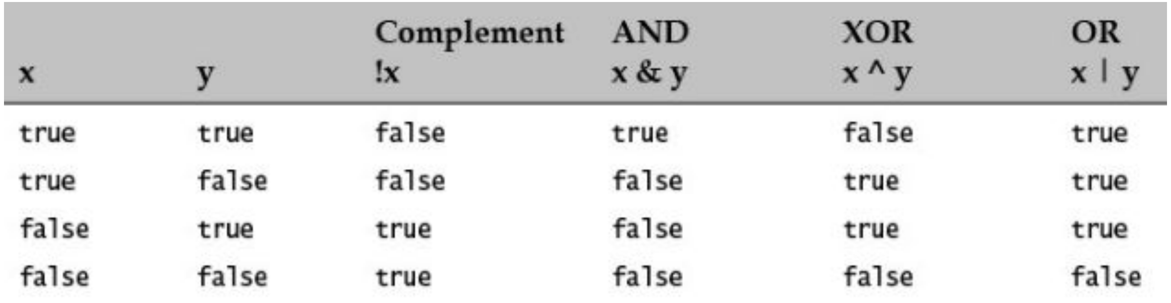


```
Pizza pizza1 = new Pizza(“VeggiesDelight”);
Pizza pizza2 = new Pizza(“VeggiesDelight”);
Pizza pizza3 = new Pizza(“CheeseDelight”);
boolean test7 = pizza1.equals(pizza2);            // false.
boolean test8 = pizza1.equals(pizza3);            // false.
boolean test9 = pizza1 == pizza2;                 // false.
pizza1 = pizza2;                                  // Creates aliases.
boolean test10 = pizza1.equals(pizza2);           // true.
boolean test11 = pizza1 == pizza2;                // true.
```

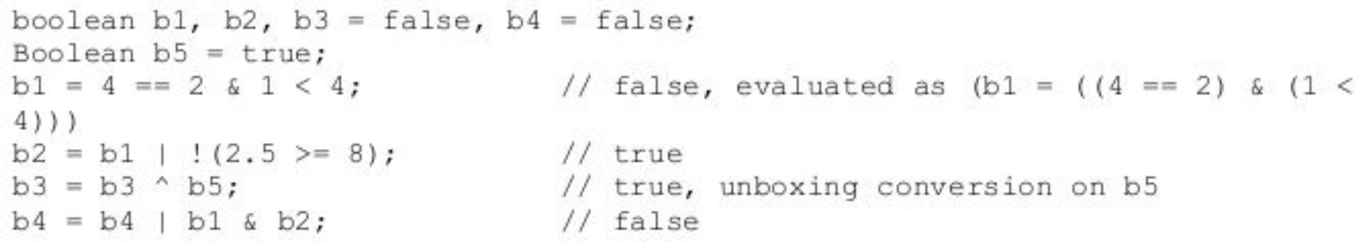
Boolean Expressions

Logical Operators: !, &, |, ^

- These operators can be applied to ***`boolean`*** or ***`Boolean`*** operands, returning a ***`boolean`*** value. Compound logical assignment operators are also defined.



x	y	Complement !x	AND x & y	XOR x ^ y	OR x | y
true	true	false	true	false	true
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

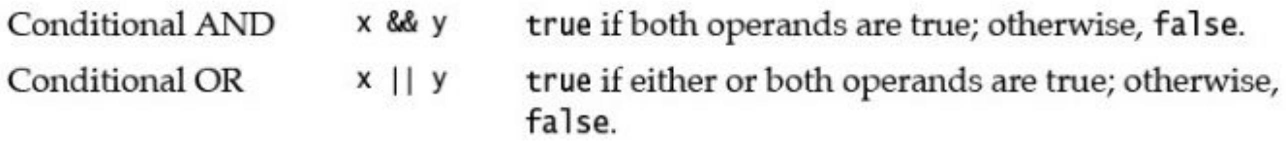


```
boolean b1, b2, b3 = false, b4 = false;
Boolean b5 = true;
b1 = 4 == 2 & 1 < 4;           // false, evaluated as (b1 = ((4 == 2) & (1 <
4)))
b2 = b1 | !(2.5 >= 8);         // true
b3 = b3 ^ b5;                  // true, unboxing conversion on b5
b4 = b4 | b1 & b2;             // false
```

Boolean Expressions

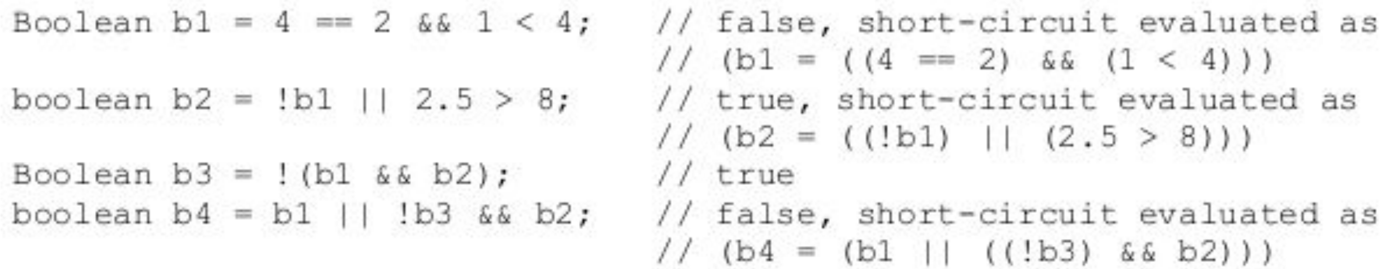
Conditional Operators: &&, ||

- The conditional operators && and || are similar to counterpart logical operators & and |.



Conditional AND	`x && y`	`true` if both operands are true; otherwise, `false`.
Conditional OR	`x || y`	`true` if either or both operands are true; otherwise, `false`.

- Unlike the logical counterparts, conditional operators' evaluation is ***short-circuited*** (i.e. if the result of the boolean expression can be determined from the left-hand operand, the right-hand operand is not evaluated).

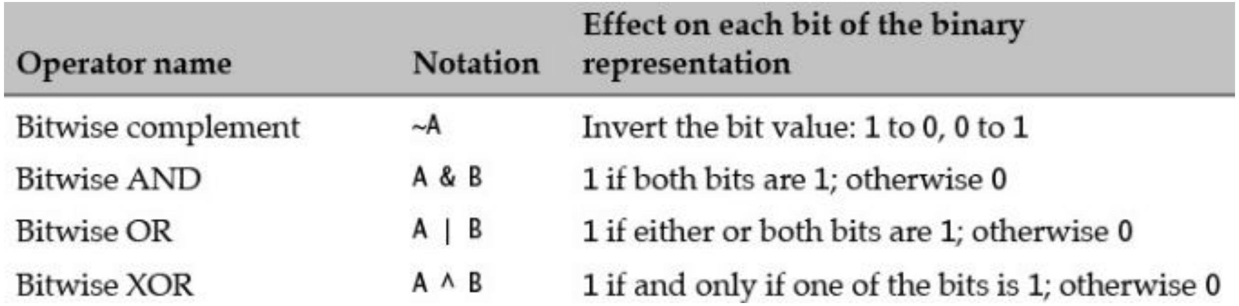


```
Boolean b1 = 4 == 2 && 1 < 4;      // false, short-circuit evaluated as
                                   // (b1 = ((4 == 2) && (1 < 4)))
boolean b2 = !b1 || 2.5 > 8;       // true, short-circuit evaluated as
                                   // (b2 = ((!b1) || (2.5 > 8)))
Boolean b3 = !(b1 && b2);          // true
boolean b4 = b1 || !b3 && b2;      // false, short-circuit evaluated as
                                   // (b4 = (b1 || ((!b3) && b2)))
```

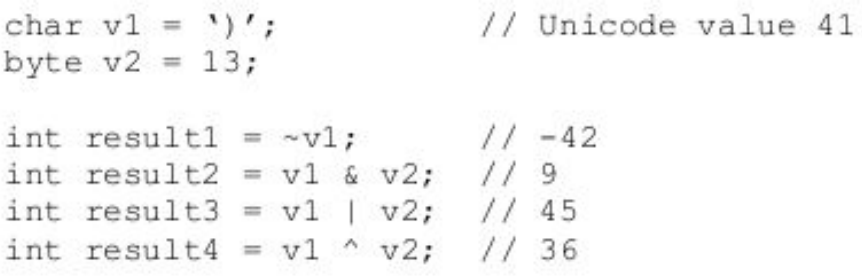
Boolean Expressions

Integer Bitwise Operators: ~, &, |, ^

- The bitwise operators perform bitwise operations between corresponding individual bit values in the operands.



Operator name	Notation	Effect on each bit of the binary representation
Bitwise complement	~A	Invert the bit value: 1 to 0, 0 to 1
Bitwise AND	A & B	1 if both bits are 1; otherwise 0
Bitwise OR	A | B	1 if either or both bits are 1; otherwise 0
Bitwise XOR	A ^ B	1 if and only if one of the bits is 1; otherwise 0

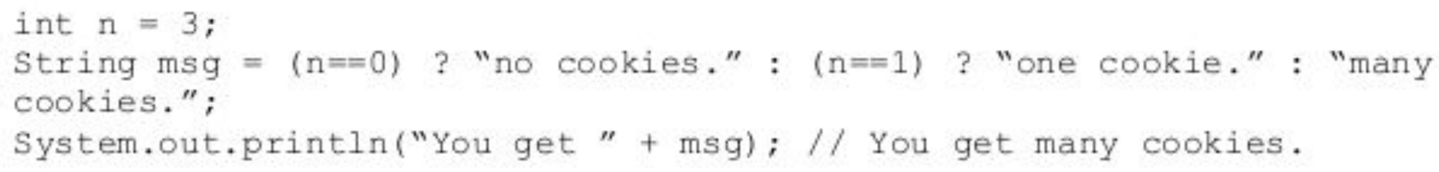


```
char v1 = ')';              // Unicode value 41
byte v2 = 13;

int result1 = ~v1;          // -42
int result2 = v1 & v2;      // 9
int result3 = v1 | v2;      // 45
int result4 = v1 ^ v2;      // 36
```

Conditional Operator

Syntax:

- **`condition ? true-expression : false-expression ;`**
- The conditional expression can be nested, and the conditional operator associates from right to left.

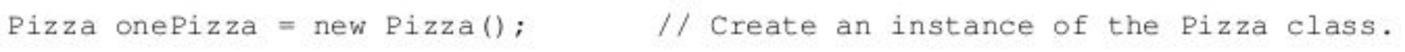


```
int n = 3;
String msg = (n==0) ? “no cookies.” : (n==1) ? “one cookie.” : “many
cookies.”;
System.out.println(“You get ” + msg); // You get many cookies.
```

Other Operators

new

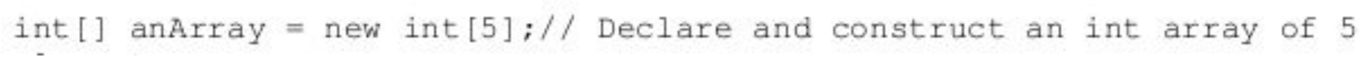
- The new operator is used to create objects, such as instances of classes (with a constructor) and arrays (with [ ] notation).



```
Pizza onePizza = new Pizza();        // Create an instance of the Pizza class.
```

[ ] notation

- This is used to declare and construct arrays, and is also to access array elements.

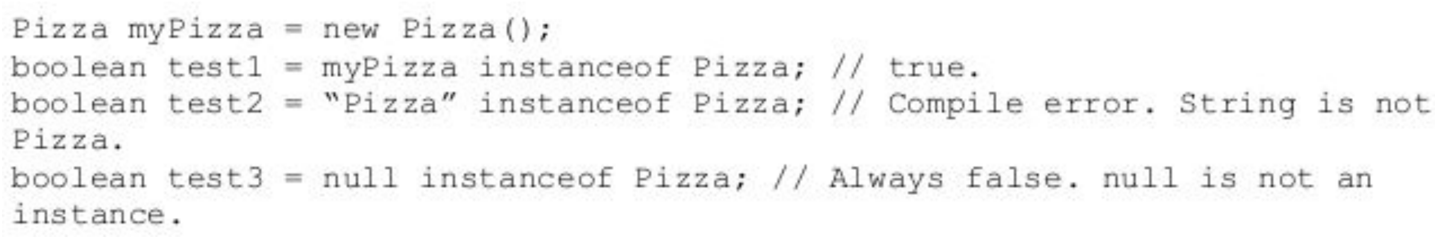


```
int[] anArray = new int[5];// Declare and construct an int array of 5
```

Other Operators

instanceof

- This boolean and binary operator is used to test the type of an object.



```
Pizza myPizza = new Pizza();
boolean test1 = myPizza instanceof Pizza; // true.
boolean test2 = “Pizza” instanceof Pizza; // Compile error. String is not
Pizza.
boolean test3 = null instanceof Pizza; // Always false. null is not an
instance.
```

IT602: Object-Oriented Programming

Control Flow Statements

Control Flow Statements

Control flow statements govern the flow of control (i.e. order of statement execution) in a program during execution.

There are three main categories of control flow statements -

- **Selection statements**: `if`, `if-else`, and `switch`
- **Iterative statements**: `while`, `do-while`, and `for`
- **Transfer statements**: `break`, `continue`, `return`, `throw`, `try-catch-finally`

*** We will cover try-catch-finally, throw in exception handling.

You’re assigned to cover selection and iterative statements on your own as they are very much similar to what you studied in C programming.

IT602: Object-Oriented Programming

Next lecture -

Declarations:
Classes and Arrays