Object-Oriented Programming

CSE703029

Faculty of Computer Science
Phenikaa University

Lecture 2: Program Control

Slides adapted from Steven Roehrig

Today We Look At

- Java operators
- Control structures
- More example programs

Java Operators

- An operator takes one or more “things” and produces a resultant “thing”.
- “Things” are usually primitive types, but they are sometimes objects.
- The “things” operated upon are called operands.
- An operator is just a function, but with a different syntax.

A Familiar Example

```
int i = 3, j = 4, k;
k = i + j;
```

- The assignment operator and the addition operator are used (each exactly once!).
- This is a more familiar syntax than, e.g.,

```
k.equals(i.add(j));
```

More Operator Facts

- All operators produce a value.
- Sometimes they produce side effects, i.e., they change the value of an operand.
- Evaluation of a statement with several operators follows precedence rules. Use parentheses for readability.
- **(x + y) * z / 3** is different than **x + y * z / 3**

Assignment Is Tricky, Part I

```
public class Number {
    public int i;
}
public class Assignment1 {
    public static void main(String[] args) {
        Number n1 = new Number();
        Number n2 = new Number();
        n1.i = 2;
        n2.i = 5;
        n1.i = n2.i;
        n2.i = 10;// what is n1.i?
    }
}
```

Assignment Is Tricky, Part II

```
public class Assignment2 {
    public static void main(String[] args) {
        Number n1 = new Number();
        Number n2 = new Number();
        n1.i = 2;
        n2.i = 5;
        n1 = n2;
        n2.i = 10;// what is n1.i?
        n1.i = 20;// what is n2.i?
    }
}
```

A Picture Might Help

Before assignment n1 = n2

reference variables

n1 n2

Number objects

n2.i n1.i

After assignment n1 = n2

reference variables

n1 n2

Number objects

n2.i n1.i

“Aliasing” In Function Calls

```
public class PassObject {
    static void f(Number m) {
        m.i = 15;
    }
    public static void main(String[] args) {
        Number n = new Number();
        n.i = 14;
        f(n);        // what is n.i now?
    }
}
```

Math Operators

- +, -, *, /, %
- Integer division truncates, i.e., 16/3 = 5
- Modulus operator returns remainder on integer division, i.e., 16%3 = 1
- Shorthand:

  x += 4; is the same as

  x = x + 4;
- This works for the other arithmetic operators as well.

Auto Increment and Decrement

- ++ increases by one, and -- decreases by one.
- Two flavors of each: pre and post:

```
int i = 1, j;

j = i++;        // j = 1, i = 2
j = ++i;        // j = 3, i = 3
j = i--;        // j = 3, i = 2
j = --i;        // j = 1, i = 1
```

Booleans and Relational Operators

- The **boolean** type has two possible values, **true** and **false**.
- The relational operators >, >=, <, <=, == and **!**= produce a **boolean** result.
- >, >=, <, <= are legal for all built-in types except **boolean**s, == and **!**= are legal for all.

Testing for (Non-)Equivalence

- The == and != operators need to be used with care with objects.

```
public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);   // prints false
                                        // String compare
        System.put.println(n1 != n2);   // prints true
    }
}
```

The **equals( )** Operator

- This exists for all *objects* (don't need it for built-in types).

```
Integer n1 = new Integer(47); //number
Integer n2 = new Integer(47);
System.out.println(n1.equals(n2);    // prints true
```

The **equals( )** Operator (cont.)

- But *exists* doesn't necessarily mean properly defined!

```
class Number {
    int i;
}
:
Number n1 = new Number();
Number n2 = new Number();
n1.i = 3; //Object
n2.i = 3;
System.out.println(n1.equals(n2));   // prints false
```

The **equals( )** Operator (cont.)

- The **equals( )** operator is properly defined for most Java library classes.
- The default behavior is to compare references, so…
- When you define a class, if you're planning to use **equals( )**, you need to define it (i.e., override the default behavior).

Logical Operators

- These are AND (**&&**), OR (||), and NOT (**!**).
- These work on **boolean**s only; if you have old "C" habits, forget them!
- Use parentheses freely to group logical expressions.
- Logical expressions *short-circuit*; as soon as the result is known, evaluation stops.

Short-Circuiting Example

```
public class ShortCircuit {
    static boolean test1(int val) {return val < 1;}
    static boolean test2(int val) {return val < 2;}
    static boolean test3(int val) {return val < 3;}
    public static void main(String[] args) {
        if (test1(0) && test2(2) && test3(2))
            System.out.println("Expression is true");
        else
            System.out.println("Expression is false");
    }
}
```

Bitwise, Shift, Ternary Operators

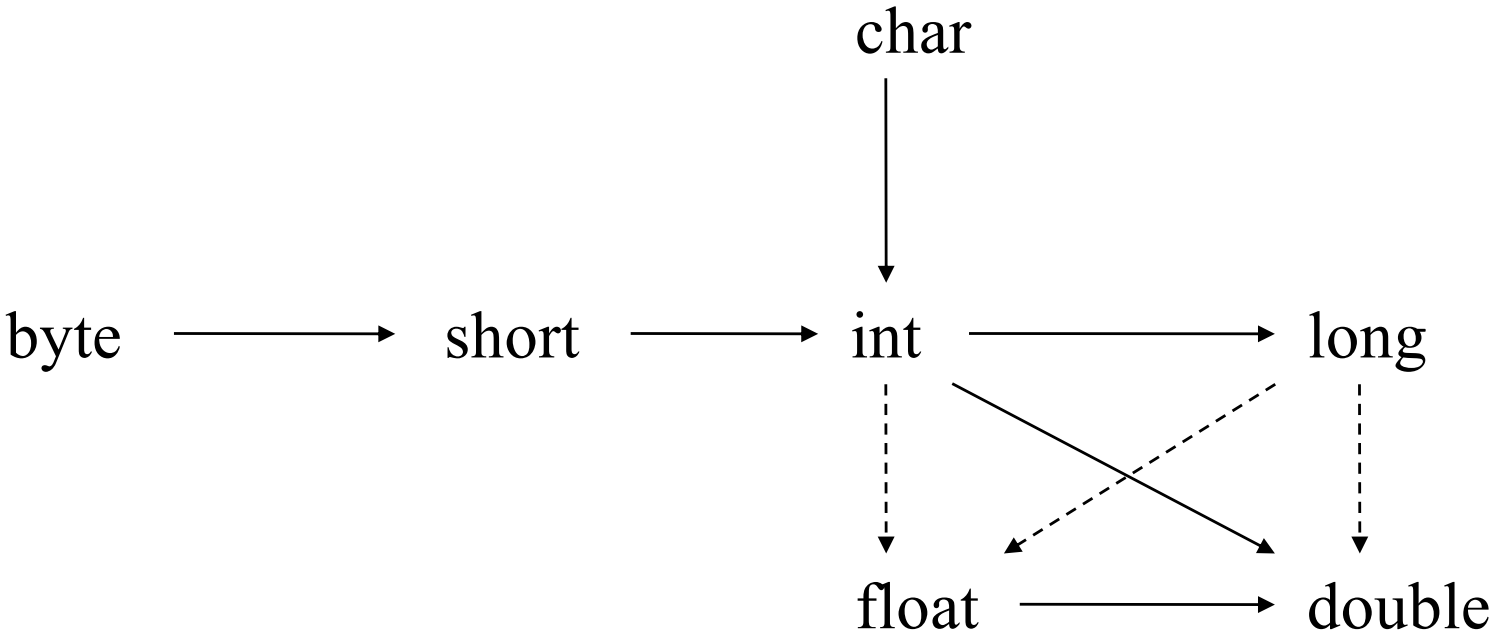
- Bitwise & shift operators manipulate individual bits in integral primitive types.
- The ternary if-else operator looks like this:
  boolean-expression ? value0 : value1
- The result is either **value0** or **value1**, depending on the truth of the **boolean**.

The **String** + Operator

- The + operator is “overloaded” for **String** objects; it means concatenation.
- It reminds me of good old C++…
- If an expression begins with a **String**, then all the following operands of + will be converted into Strings:

```
int x = 0, y = 1, z = 2;
String myString = “x, y, z ”;
System.out.println(myString + x + y + z);
```

A Useful Figure



When multiple types are mixed in an expression, compiler will convert the operands into the **higher type**:
double, float, long, int

Casting

- A cast produces a temporary new value of a designated type.
- Implicit and explicit casts:

```
int i = 2;
double d= i;// OK, since d can hold all of i
float g = 3.14159F;
//! int j = g;          // not OK, loses information
// int is smaller [memory] type than float
//double is bigger [memory] type than int
int k = (int) g;       // OK, compiler is reassured
                       // forced type
```

Execution Control: i**f-else**

```
if (boolean_expression)
    statement
else if(boolean_expression)
    statement
:
else if(boolean_expression)
    statement
else
    statement
```

if-else Example

```
public int test(int testVal, int target) {
    int result = 0;
    if (testVal > target)
        result = 1;
    else if (testVal < target)
        result = -1;
    else {
        System.out.println("They are equal");
        result = 0;
    }
    return result;
}
```

Execution Control: **return**

- Exit a method, returning an actual value or object, or not (if the return type is void).

```
public int test(int testVal, int target) {
if (testVal > target)
        return 1;//return int
else if (testVal < target)
        return  -1; //return int

else {
        System.out.println("They are equal");
        return 0;//return int

   }
```

Three Kinds of **Iteration**

while **(boolean_expression)** **// evaluate first**

statement_or_block

do

statement_or_block **// evaluate last**

while **(boolean_expression)**

for **(initialization ; boolean_expression ; step)**

statement_or_block

Example:

```
for (int i = 0; i < myArray.size(); i++) {
    myArray[i] = 0;
}
```

Break and Continue

```
public class BreakAndContinue {
    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            if (i == 74) break;              // out of for loop
            if (i % 9 != 0) continue; // next iteration
            System.out.println(i);
        }
    }
}
```

Selection Via **switch**

```
for (int i = 0; i < 100; i++) {
    char c = (char) (Math.random() * 26 + ‘a’);
    switch(c) {
    case ‘a’:
    case ‘e’:
    case ‘i’:
    case ‘o’:
    case ‘u’:
        System.out.println(“Vowel”); break;
    case ‘y’:
    case ‘w’:
        System.out.println(“Sometimes a vowel”); break;
    default:
        System.out.println(“Not a vowel”);
}
```

Digression on Random Numbers

- Despite theory, most random number generators are more like "kids playing with matches".
  - See "Random Number Generators: Good Ones Are Hard to Find" by S. Park and K. Miller, *CACM* Oct. 1988.
- Most random number generators use "multiplicative linear congruential" schemes:
  - A modulus **m**, a large prime integer
  - A multiplier **a**, an integer in the range 2,3,…m-1
  - These produce a sequence z_1, z_2, z_3… using the iterative equation
  - $z_{i+1} = f(z_i) = a*z \% m$

Random Numbers (cont.)

- The sequence is initiated by choosing a seed.
- Example: f(z) = 6z %13. This produces the sequence ...1 , 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1,...
- Example: f(z) = 7z % 13. This produces the sequence ...1 , 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1,...
- Is the latter somehow "less random"?
- Example: f(z) = 5z %13. This produces the sequence ...1 , 5, 12, 8, 1...

Using Java's RNGs (cont.)

- java.lang.Math
  - static double random() random in [0, 1.0)
- The sequence doesn't seem to be repeatable
- Bad for debugging
- Good for experimental work

Using Java's RNGs (cont.)

- java.util.Random
  - Constructors:
    - **Random()**
    - **Random(long seed)**
  - Methods:
    - nextInt() random in $(-2^{31}, 2^{31}-1)$
    - nextInt(int n) random in [0, n)
    - nextFloat() random in [0, 1)
    - setSeed(long seed)
- java.lang.Math
  - static double random() random in [0, 1.0)