Object Oriented Software Design

Introduction to Java - II

Giuseppe Lipari
`http://retis.sssup.it/~lipari`

Scuola Superiore Sant’Anna – Pisa

October 28, 2010

Outline

Outline

Makefile

- All examples in these slides are included in directories `material/examples/<lecture_name>`
- To compile them, all you have to do is to enter the directory with a terminal, and type `make`
  - This will execute the make command which interprets file `makefile`
  - The file contains instructions on how to compile and execute the programs
- To execute a program, type `make <ProgramName>`

Bruce Eckel examples

- Some of the examples are taken by Bruce Eckel's "Thinking in Java"
  - These examples use utility classes that can be found in directory `com/bruceeckel/`
  - So, you will need this directory to compile and execute Bruce Eckel's examples
- How to install examples and slides:
  - Download from my web site `http://retis.sssup.it/~lipari` the two zip files `material.zip` and `bruceeckel.zip`
  - unzip them in the same directory
  - Now you are ready to compile and execute the code

Testing

- In some of the examples, Bruce Eckel uses a `Test` class to check the correctness of the example
    - The `Test` class compares the output of the program with the expected output
        - If they are the same, the program is correct
        - If they differ, the program is interrupted with an error code
    - We will see in a while how it works

Outline

Operators in JAVA

- An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary method calls, but the effect is the same
- Addition (+), subtraction and unary minus (-), multiplication (*), division (/), and assignment (=) all work much the same in any programming language
- Almost all operators work only with primitives (cannot be used on references)
  - **Exceptions** are '=', '==' and '!=', which work with all objects (and are a point of confusion for objects).
  - In addition, the `String` class supports '+' and '+='.

Precedence

- Operator precedence defines how an expression evaluates when several operators are present
- JAVA has specific rules that determine the order of evaluation
- Programmers often forget the other precedence rules, so you should use parentheses to make the order of evaluation explicit.
- Example:

```
a = x + y - 2/2 + z;
```

```
a = x + (y - 2)/(2 + z);
```

Assignment

- Assignment is performed with the operator **=**
- It means "take the value of the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*)."
  - An *rvalue* is any constant, variable, or expression that can produce a value,
  - an *lvalue* must be a distinct, named variable.
- Assignment of primitives is quite straightforward
  - For example, if you say `a = b` for primitives, then the contents of `b` are copied into `a`
  - If you then go on to modify `a`, `b` is naturally unaffected by this modification

Assignment of objects

- Whenever you manipulate an object, what you're manipulating is the reference
  - if you say `c = d` for objects, you end up with both c and d referencing the object that, originally, only d pointed to
- Example: `./examples/04.java-examples/Assignment.java`

Aliasing

- This phenomenon is often called aliasing, and it's a fundamental way that Java works with objects.
- But what if you don't want aliasing to occur in this case?
    - You have to copy members one by one
- Aliasing happens also when passing objects to methods
    - In other languages, we would say that an object is passed by reference, instead of passing it by value (in C++ it is possible to choose)
    - In JAVA, all you manage is a reference, so you pass references by value (and objects by reference)
    - Example: `./examples/04.java-examples/PassObject.java`

Mathematical Operators

- An example is here:
  `./examples/04.java-examples/MathOps.java`
- Notice the use of the regular expressions to check the correctness of the output:
  - `"-?\\d+"` means a number that may be preceded by a minus sign (`"-?"`) and may have an arbitrary number of digits (`"\\d+"`)
  - `"%%"` means that the string is actually a regular expression

Increment/Decrement

- Something you will need to do very often is to increment/decrement integer variables
  - For example, for implementing cycles
- As in C, you can simply use pre- and post- increment/decrement operators `++` and `--`
  - With pre-increment, the result of the expression is the value of the variable after the increment (as in `++i`)
  - With post-increment, the result of the expression is the value of the variable before it is incremented
- Example: `./examples/04.java-examples/AutoInc.java`
- Pay attention to the difference between pre- and post- increment!

Comparison

- Often, you may want to compare things for equality
- Relational operators (`<`, `>`, `==`, `<=`, `>=`, `!=`), returns a boolean
  - They can be used for primitive types, with their obvious meaning
  - Operators `==`, `!=` also work with reference,
    - **WARNING**: they test equality of **references**!
  - Example:
    `./examples/04.java-examples/Equivalence.java`

Equality of objects

- What if I want to test equality of objects?
  - You have to use method `equals()`
  - All classes implicitely derive from a single standard class called `Object`, for which this method is defined
  - All classes *inherit* such a method, thus you can call `equals()` on every class
  - By default, `equals()` compares references (as ==)
  - However, many classes redefine (*override*) the method to implement a different behavior (for example, `Integer` reimplement `equals` to compare the content of the objects)

Example of equals()

- An example of `equals()` for `Integer`
  - `./examples/04.java-examples/EqualsMethod.java`
- An example of `equals()` for a user defined class
  - `./examples/04.java-examples/EqualsMethod2.java`
  - Here the behaviour is the same as ==

Logical Operators

- Logical operators take boolean values and return a boolean
    - `&&, ||, !`
- Unlike C and C++, logical operators cannot be applied to non-boolean values
- See `./examples/04.java-examples/Bool.java`

Side effects

- A boolean expression will be evaluated only until the truth or falsehood of the entire expression can be unambiguously determined
- As a result, the latter parts of a logical expression might not be evaluated
- `./examples/04.java-examples/ShortCircuit.java`
- Pay attention to side-effects of expressions!

Bitwise and shift operators

- The bitwise operators allow you to manipulate individual bits in an integral primitive data type
  - `&`, `|`, `^` respectively and, or and xor
  - ~ is the bitwise not (takes only one operand)
- The shift operators allow to shift the binary representation of an integer to the left (<<) or to the right (>>)

```
int i = -1;
i = i << 10;  // signed shift ten bits to the right
i = i >>> 5;  // unsigned shift 5 bits to the left
```

- Example: `./examples/04.java-examples/URShift.java`
- They come from the fact that JAVA was used for embedded systems
- Not very much used, read the manual (or “Thinking in JAVA”) for more information

Ternary if-else operator

- This operator has the form:

```
boolean-exp ? exp0 : exp1
```

- If the boolean expression is true, `exp0` is evaluated, else `exp1` is evaluated
  - Of course, they must have the same type
- An example is the following:

```
static int ternary(int i) {
  return i < 10 ? i * 100 : i * 10;
}
```

- The above code is equivalent to:

```
static int alternative(int i) {
  if (i < 10) return i * 100;
  else        return i * 10;
}
```

String + operator

- The + operator in JAVA, when applied to `Strings`, concatenates the two strings
  - In C++, it is possible to redefine (override or overload) the use of any operator on any class
  - JAVA has not concept of overloaded operators, like in C++, this is the only exception and it is built-in
  - The reason was to maintain simplicity
- If one of the two operands is a String, the other is converted into a String before being concatenated

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

Outline

Casting

- The word cast is used in the sense of “casting into a mold.”
- JAVA will automatically change one type of data into another when appropriate (*implicit casting*)
  - For example, an **`int`** into a **`float`** (no loss of precision)
- You can make casting *explitic* by using appropriate casting operators

```
int i = 200;
long l = (long)i;
long l2 = (long)200;
```

  - (In both cases, the casting is superflous)
- JAVA lets you cast every primitive type into every other primitive type, except for boolean (which cannot be casted at all)
- In some cases there will be loss of precision (truncation)

Various

- Literals:
  - If you want to specify the type of a nuerical constant, you can put a character after the number to specify the type: `L` stands for **`long`**, `F` stands for **`float`**, and `D` stands for **`double`**,
- Casting:
  - In general, the largest data type in an expression is the one that determines the size of the result of that expression; if you multiply a float and a double, the result will be double; if you add an int and a long, the result will be long.
- JAVA has no sizeof()
  - In C/C++, different data types might be different sizes on different machines, so the programmer must find out how big those types are when performing operations that are sensitive to size
  - JAVA programs are portable, hence there is no sizeof() operator

Overflow

- JAVA does not check for overflow (it would be too heavy)
- Pay attention to not overflow numerical values after an arithmetic expression!
- Example: `./examples/04.java-examples/Overflow.java`

Outline

if-else

- JAVA support all control flow statements of C/C++, except for **`goto`**

if-else

- JAVA support all control flow statements of C/C++, except for **`goto`**
- All conditional statements use the truth or falsehood of a conditional expression to determine the execution path

if-else

- JAVA support all control flow statements of C/C++, except for **`goto`**
- All conditional statements use the truth or falsehood of a conditional expression to determine the execution path
- the **if-else** is the most basic form of flow control

```
if(Boolean-expression)
  statement
```

or

```
if(Boolean-expression)
  statement
else
  statement
```

if-else

- JAVA support all control flow statements of C/C++, except for **`goto`**
- All conditional statements use the truth or falsehood of a conditional expression to determine the execution path
- the **if-else** is the most basic form of flow control

```
if(Boolean-expression)
  statement
```

or

```
if(Boolean-expression)
  statement
else
  statement
```

- *statement* can be a simple expression, a statement, or a block of statements grouped by curly braces

Example

IfElse.java

```
public class IfElse {
    static Test monitor = new Test();
    static int test(int testval, int target) {
        int result = 0;
        if (testval > target)
            result = +1;
        else if (testval < target)
            result = -1;
        else
            result = 0; // Match
        return result;
    }
    public static void main(String[] args) {
        System.out.println(test(10, 5));
        System.out.println(test(5, 10));
        System.out.println(test(5, 5));
```

While

- To loop a number of times over a set of statements, you can use the *while* statement:

```
while(Boolean-expression)
  statement
```

- Example: While.java

```
public class While {
    static Test monitor = new Test();
    static public void main(String args[]) {
        int i = 0;
        while (i<5) {
            System.out.println(i + ": " + (i*2));
            i++;
        }
```

- Bruce's Eckel example:
  `./examples/04.java-examples/WhileTest.java`

do-while

- The *do-while* is pretty similar to the *while* statement:

```
do
  statement
while(Boolean-expression);
```

- Remember:
  - the *do-while* statement always execute at least once
  - the *while* statement could never execute

for loop

- A for loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of “stepping.”

```
for(initialization; Boolean-expression; step)
  statement
```

- Any of the expressions initialization, Boolean-expression or step can be empty
  - An infinite loop: **`for(;;);`**
- It works like this:
  - The expression is tested before each iteration,
  - as soon as it evaluates to false, execution will continue at the line following the for statement.
  - At the end of each loop, the step executes.

Example

ListCharacters.java

```
public class ListCharacters {
  static Test monitor = new Test();
  public static void main(String[] args) {
    for(int i = 0; i < 128; i++)
      if(Character.isLowerCase((char)i))
        System.out.println("value: " + i +
          " character: " + (char)i);
```

- Note that the variable i is defined at the point where it is used, inside the control expression of the for loop, rather than at the beginning of the block denoted by the open curly brace.
- The scope of i is the expression controlled by the for.

Comma operator

- You can define multiple variables within a for statement, but they must be of the same type
- You can also put multiple statements separated by a comma inside the step, or initialization parts

CommaOperator.java

```
public class CommaOperator {
  static Test monitor = new Test();
  public static void main(String[] args) {
    for(int i = 1, j = i + 10; i < 5;
        i++, j = i * 2) {
      System.out.println("i= " + i + " j= " + j);
    }
```

- You can see that in both the initialization and step portions, the statements are evaluated in sequential order. Also, the initialization portion can have any number of definitions of one type

Break and Continue

- **break** quits the loop without executing the rest of the statements
- **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration

BreakAndContinue.java

```
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);
        }
        int i = 0;
        // An "infinite loop":
        while(true) {
            i++;
            int j = i * 27;
            if(j == 1269) break; // Out of loop
            if(i % 10 != 0) continue; // Top of loop
            System.out.println(i);
        }
```

Goto?

- We said that JAVA has no goto statement

Goto?

- We said that JAVA has no goto statement
- Actually, JAVA has labels that can be used in conjunction with *break* and *continue*

Goto?

- We said that JAVA has no goto statement
- Actually, JAVA has labels that can be used in conjunction with *break* and *continue*
- A label is identified by a name followed by a colon:

```
mylabel:
```

Goto?

- We said that JAVA has no goto statement
- Actually, JAVA has labels that can be used in conjunction with *break* and *continue*
- A label is identified by a name followed by a colon:

```
mylabel:
```

- The only place a label is useful in Java is **right before** an iteration statement

Goto?

- We said that JAVA has no goto statement
- Actually, JAVA has labels that can be used in conjunction with *break* and *continue*
- A label is identified by a name followed by a colon:

```
mylabel:
```

- The only place a label is useful in Java is **right before** an iteration statement
  - It is used when you have multiple nested loops and you want to go out from all of them at once

Label

- Let's see how it works

```
label1:
outer-iteration {
  inner-iteration {
    //...
    break; // 1
    //...
    continue;  // 2
    //...
    continue label1; // 3
    //...
    break label1;  // 4
  }
}
```

- In case 1, the break breaks out of the inner iteration and you end up in the outer iteration

Label

- Let's see how it works

```
label1:
outer-iteration {
  inner-iteration {
    //...
    break; // 1
    //...
    continue;  // 2
    //...
    continue label1; // 3
    //...
    break label1;  // 4
  }
}
```

- In case 2, the continue moves back to the beginning of the inner iteration

Label

- Let’s see how it works

```
label1:
outer-iteration {
  inner-iteration {
    //...
    break; // 1
    //...
    continue;  // 2
    //...
    continue label1; // 3
    //...
    break label1;  // 4
  }
}
```

- in case 3, the continue label1 breaks out of the inner iteration and the outer iteration, all the way back to label1
  - Then it does in fact continue the iteration, but starting at the outer iteration

Label

- Let’s see how it works

```
label1:
outer-iteration {
  inner-iteration {
    //...
    break; // 1
    //...
    continue;  // 2
    //...
    continue label1; // 3
    //...
    break label1;  // 4
  }
}
```

- In case 4, the break label1 also breaks all the way out to label1, but it does not reenter the iteration.

Examples

- Example with for loop:
  `./examples/04.java-examples/LabeledFor.java`
- Example with while loop:
  `./examples/04.java-examples/LabeledWhile.java`

switch

- The switch statement has the following form:

```
 switch(integral-selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; break;
  case integral-value3 : statement; break;
  case integral-value4 : statement; break;
  case integral-value5 : statement; break;
  // ...
  default: statement;
}
```

- Each case ends with a break, which causes execution to jump to the end of the switch body
- This is the conventional way to build a switch statement, but the break is optional.
- If it is missing, the code for the following case statements execute until a break is encountered
- Example:
  `./examples/04.java-examples/VowelsAndConsonants.ja`

Exercises

Exercise

Write a program that generates 25 random int values. For each value, use an if-else statement to classify it as greater than, less than, or equal to a second randomly-generated value.

Exercise

Write a program that uses two nested for loops and the modulus operator (%) to detect and print prime numbers (integral numbers that are not evenly divisible by any other numbers except for themselves and 1) Test the correctness of the program on the first 10 primes using the Test class