Java Programming AP Edition
U2C5 Loops

FINAL WORDS I (BREAK LEVELS AND OTHER TOPICS ABOUT LOOPS)

ERIC Y. CHOU, PH.D. IEEE SENIOR MEMBER

(1) Break Levels (TestBreak.java)

for iteration/loop/method/program


```
pass:  // Java does not have pass statement;
    { /* put nothing here */
    }
continue:  // break from an iteration
     for (i=1; i<10; i++)  {
            if (i==3) continue;
            System.out.print(i);
     } /* 1, 2, 4, 5, 6, 7, 8, 9 will be printed */
```

(1) Break Levels

for iteration/loop/method/program

break: // breaking from a **loop**

```
for (i=1; i<10; i++)  {  /* 1, 2 will be printed */
      if (i==3) break;
      System.out.print(i);
}
```

return: // breaking from a **method**

```
void f() {/* nothing done in this function */
      return;
      System.out.print("Here !");
 }
```

exit: // breaking from a **program**.

```
System.exit(0);  /* program terminated */
```

continue, **break**, **return** are keywords in Java. **exit** is a method in System package. Java has no **pass**.

(2) Minimizing Numerical Errors (TestSum.java)

Numeric errors involving floating-point numbers are inevitable. This section discusses how to minimize such errors through an example.

Here is an example that sums a series that starts with 0.01 and ends with 1.0. The numbers in the series will increment by 0.01, as follows: **0.01 + 0.02 + 0.03 + … + 0.99 + 1.0**

0.01 + 0.02 + 0.03 + … + 0.99 + 1.0 (better)

1.0 + 0.99 + … + 0.03 + 0.02 + 0.01 (worse)

(2) Minimizing Numerical Errors for-loop with double


```
double sum;

for (double x = 0.01f ; x != 1.0; x += 0.01f) {
      sum += x;
}

System.out.println(“Sum: ”, x);
```

(2) Minimizing Numerical Errors for-loop with double


```
double sum;  // x <= 1.0 won't work

for (double x = 0.01f ; x < 1.005; x += 0.01f) {
      sum += x;
}
  // increment by 0.01f, it won't accumulate
  // more than 0.005 for 100 additions of 0.01f

System.out.println("Sum: ", x);
```

(2) Minimizing Numerical Errors for-loop with double


```
double sum;

for (int i=1 ; i <=100; i++) { // use integer-indexed loop
      sum += 0.01f * i;
}

System.out.println("Sum: ", x);
```

(3) Sentinel-Controlled loop

Controlling a Loop with a Sentinel Value (SentinelValue.java)

Another common techniques for controlling a loop is to designate a special value when reading and processing a set of values.

This special input value, known as a **sentinel value**, signifies the end of the input. A loop that uses a sentinel value to control its execution is called a **sentinel-controlled loop**.

// I personally prefer a **controlling flag** instead of **sentinel value**.

(4) Input and Output Redirection

In windows system or Unix system, you may redirect your input stream from console to file or from file to console/printer. In window system, console is short for **con**, while printer is short for **prn**.

You can store the data separated by whitespaces in a text file, say input.txt, and run the program using the following command (on windows command line):

```
java SentinelValue < input.txt
```

(4) Input and Output Redirection

This command is called input redirection. The program takes the input from the file input.txt rather than having the user type the data from the keyboard at run-time. Suppose that the contents of the file are

2 3 4 5 6 7 8 9 12 23 32

23 45 67 89 92 12 34 35 3 1 2 4 0

The program should get sum to be 518.

(4) Input and Output Redirection

Similarly, there is output redirection, which sends the output to a file rather than displaying it on the console. The command for output redirection is:

```
java ClassName > output.txt
```

Input and Output redirection can be used in the same command. For example, the following command gets input from input.txt and sends output to output.txt.

```
java SentinelValue < input.txt > output.txt
```