Introduction to Object-Oriented Programming

Loops

Christopher Simpkins
chris.simpkins@gatech.edu
Loops and Iteration

Algorithms often call for repeated action or iteration, e.g.

- “repeat ... while (or until) some condition is true” (looping) or
- “for each element of this array/list/etc. ...” (iteration)

Java provides three iteration structures:

- while loop
- do-while loop
- for iteration statement
while and do-while

while loops are pre-test loops: the loop condition is tested before the loop body is executed

```
while (condition) {
    // condition is any boolean expression
    // loop body executes as long as condition is true
}
```

do-while loops are post-test loops: the loop condition is tested after the loop body is executed

```
do {
    // loop body executes as long as condition is true
} while (condition)
```

The body of a do-while loop will always execute at least once.
The general `for` statement syntax is:

```java
for(initializer; condition; update) {
    // body executed as long as condition is true
}
```

- **`initializer`** is a statement
  - Use this statement to initialize value of the loop variable(s)

- **`condition`** is tested before executing the loop body to determine whether the loop body should be executed. When false, the loop exits just like a while loop

- **`update`** is a statement
  - Use this statement to update the value of the loop variable(s) so that the `condition` converges to `false`
for Statement vs. while Statement

The for statement:

```java
for(int i = 0; i < 10; i++) {
    // body executed as long as condition is true
}
```

is equivalent to:

```java
int i = 0
while (i < 10) {
    // body
    i++; // increment i
}
```

for is Java’s primary iteration structure. In the future we’ll see generalized versions, but for now for statements are used primarily to iterate through the indexes of data structures and to repeat code a particular number of times.
for Statement Examples

Here’s an example from CharCount.java. We use the for loop’s index variable to visit each character in a String and count the digits and letters:

```java
int digitCount = 0, letterCount = 0;
for (int i = 0; i < input.length(); ++i) {
    char c = input.charAt(i);
    if (Character.isDigit(c)) digitCount++;
    if (Character.isAlphabetic(c)) letterCount++;
}
```

And here’s a simple example of repeating an action a fixed number of times:

```java
for (int i = 0; i < 10; ++i)
    System.out.println("Meow!");
```
for Statement Subtleties

Better to declare loop index in `for` to limit it’s scope. Prefer:

```java
for (int i = 0; i < 10; ++i)
```

to:

```java
int i; // Bad. Loop index variable visible outside loop.
for (i = 0; i < 10; ++i)
```

You can have multiple loop indexes separated by commas:

```java
String mystery = "mnerigpaba", solved = ""; int len = mystery.length();
for (int i = 0, j = len - 1; i < len/2; ++i, --j) {
    solved = solved + mystery.charAt(i) + mystery.charAt(j);
}
```

Note that the loop above is one loop, not nested loops.

Beware of common “extra semicolon” syntax error:

```java
for (int i = 0; i < 10; ++i); // oops! semicolon ends the statement
print(meow); // this will only execute once, not 10 times
```
Forever

Note that in the context of programming, infinite means “as long as the program is running.” Here are two idioms for infinite loops. First with for:

```java
for (;;) {
    // ever
}
```

and with while:

```java
while (true) {
    // forever
}
```

Infinite loops are useful for things like game loops and operating system routines that poll input buffers or wait for incoming network connections. In both of these cases the loop is intended to run for the duration of the program. See Loops.java for loop examples.
Java provides two non-structured-programming ways to alter loop control flow:

- **break** exits the loop, possibly to a labeled location in the program
- **continue** skips the remainder of a loop body and continues with the next iteration

Consider the following while loop:

```java
boolean shouldContinue = true;
while (shouldContinue) {
    System.out.println("Enter some input (exit to quit): ");
    String input = System.console().readLine();
    doSomethingWithInput(input); // We do something with "exit" too.
    shouldContinue = (input.equalsIgnoreCase("exit")) ? false : true;
}
```

We don’t test for the termination sentinel, “exit,” until after we do something with it. Situations like these often tempt us to use **break**...
We could test for the sentinal and break before processing:

```java
boolean shouldContinue = true;
while (shouldContinue) {
    System.out.println("Enter some input (exit to quit): ");
    String input = System.console().readLine();
    if (input.equalsIgnoreCase("exit")) break;
    doSomethingWithInput(input);
}
```

But it’s better to use structured programming:

```java
boolean shouldContinue = true;
while (shouldContinue) {
    System.out.println("Enter some input (exit to quit): ");
    String input = System.console().readLine();
    if (input.equalsIgnoreCase("exit")) {
        shouldContinue = false;
    } else {
        doSomethingWithInput(input);
    }
}
```
Write a program that determines whether a String is a palindrome. A palindrome is a string of characters that reads the same when reversed, e.g. “radar”.

- Your program should use the first command line argument as the String to test
- You should ignore case, e.g. ‘R’ == ’r’
- You should ignore spaces, e.g. “a but tuba” is a palindrome
- Try to do this with a single for loop