CMSC 131: Chapter 10 (Supplement)
Interfaces and Flow

Motivation

Two Opposing Goals that Java programmers must deal with:

**Strong Typing:** In strongly typed languages, like Java, the type of every variable must be specified.

**General-Purpose Functions:** We would like to write methods that can be applied to many different types.

Example: Sorting involves taking a list of elements and arranging them in increasing order. We would like to be able to sort lists of ints, doubles, Strings, Dates, Rationals, etc.

The Problem: Strong typing implies that to write a sorting function, we need to specify the types of the parameters (int, double, String, etc.). This makes it impossible to write a generic sorting function.

It is harder to debug and maintain many copies of the same method.

Java Interfaces

**Java Interface:** Java supports a language construct called “interface” which allows you to write general purpose functions.

**How it works:** Suppose you want to write a sorting method for objects of some class X.

- You implement a **general-purpose sorting method**, using a comparison method (e.g., compareTo( )).

- The user of your sorting function defines this comparison method (compareTo( )) for objects of class X.

- Now it is possible to invoke your general sorting method on objects of class X.

**To make this work:** Java needs to provide some mechanism for general-purpose functions (like sort) to specify what behavior they require from specific classes (like X).
Java Interfaces (continued)

A Java Interface is a formal way for a class to promise to implement certain methods. We say that a class implements an interface if it provides these methods.

Interface:
- Is defined by the keyword interface (rather than class)
- It defines methods, but does not provide a method body (the executable statements that make up the method).

Defining a Java Interface:
- A Java interface is collection of method declarations.
- These declarations are abstract, which means that we do not supply the body of the method.

```java
public interface Y {
    public void someMethod(int z);
    public int anotherMethod();
}
```
- These methods are usually public, since they are expected to be part of an object's public interface.
- Notice that an interface is not a class. For example, you cannot create an instance using "new Y".

Implementing an Interface

Implementing Java Interface:
- A class is said to "implement" an interface if it provides definitions for these methods.
- To inform Java that a class implements a particular interface Y, we add "implements Y" after the class name:

```java
public class X implements Y {
    // ...(instance data and other methods)...
    public void someMethod(int z) { /* give implementation here */ }
    public int anotherMethod() { /* give implementation here */ }
}
```
- Now, we may use an X any place that an object of type Y is expected.
Review of Control Flow

Control Flow: Controls the order in which statements are executed in your program

if and if-else: Conditionally execute a block of statements based on a boolean conditional expression. Example:

```java
if (option == 1)
    System.out.println("Read image");
else if (option == 2)
    System.out.println("Double");
else if (option == 9)
    System.out.println("Quit");
else
    System.out.println("Sorry, invalid");
```

while and do-while loops: Repeatedly execute some block of statements as long as a condition holds.

The Switch Statement

Switch Statement: is a convenient (and often more efficient) way to perform a multi-way conditional based on a single control value.

Example:

```java
switch (option) {
    case 1:
        System.out.println("Read image");
        break;
    case 2:
        System.out.println("Double");
        break;
    case 9:
        System.out.println("Quit");
        break;
    default:
        System.out.println("Sorry, invalid");
        break;
}
```
The Switch Statement

General form:

```java
switch ((control-expression) ){
  case (case-label-1) :
    (statement-sequence-1)
    break;
  case (case-label-2) :
    (statement-sequence-2)
    break;
...
  case (case-label-n) :
    (statement-sequence-n)
    break;
  default :
    (default-statement-sequence)
    break;
}
```

The Switch Statement

The control expression can be of one of the following types:
  char, int, short, byte.

  • not float or double,
  • not boolean or long
  • not an object (Too bad! Strings would have been nice.)

The "break" statement jumps out of the switch statement. Otherwise control flow just "falls through" into the next case.

```java
int option = 2;
switch ( option ) {
  case 1:
    System.out.println( "Read image" );
  case 2:
    System.out.println( "Double" );
  case 9:
    System.out.println( "Quit" );
  default:
    System.out.println( "Sorry, invalid" );
}
```
The Switch Statement

The falling through behavior is handy, because it allows you to combine cases. Example: Allowing either upper-case or lower-case for characters:

```java
char command = 'D';
switch (command) {
    case 'I':
    case 'I':
        MyUtility.insert();
        numberOfItems++;
        break;
    case 'd':
    case 'D':
        MyUtility.delete();
        numberOfItems--;
        break;
}
...
```

The Switch Statement

The "default" case is optional. If it is not included, and no case matches, then the switch statement does nothing.

It is considered good practice to always include a default case, if only to print an error message of an illegal choice.

Cases are not required to be in order. The following is legal, but is confusing for the reader.

```java
switch (option) {
    case 2:
    ...
    case 9:
    ...
    default:
    ...
    case 1:
}
```

Recommended: List cases in increasing order, and put the default last.
The For Loop

Common loop structure:

(initialization)
while ( (boolean-test) ) {
    (loop-body)
    (update)
}

sum = 0.0;
while ( n <= 3 ){
    sum += n;
    n++;
}

The for loop provides a shorthand for expressing this type of loop:

for ( (initialization); (boolean-test); (update) ) {
    (loop-body)
}

The above loop is equivalent to:

sum = 0.0;
for ( n = 1; n <= 3; n++ ){
    sum += n;
}

The For Loop: Examples

Loop that counts from 100 down to 0: (100, 99, 98, ..., 1, 0)

for ( count = 100; count >= 0; count-- )
    System.out.println( count + " bottles of beer on the wall" );

Loop that counts by twos from 0 up to max+10: (0, 2, 4, ..., max+10)

for ( m = 0; m <= max+10; m += 2 ){
    // ... something exciting ...
}

It is very convenient to declare the loop-control variable within the initialization. The scope of the variable is limited to the for loop:

for ( int i = 0; i < 20; i++ ){
    sum = sum + i;
}
System.out.println( i );               // this is a compiler error: i only accessible inside the loop
The For Loop: Further Elements

Multiple Initialization/Increment: Sometimes it is useful to have multiple initializations and multiple increments. This can be achieved by separating the operations by commas.

Example:

```java
for (m = 0; n = 100; m < n; m++, n -= 2 ){
    // ... something senseless ...
}
```
Equivalent to:

```java
m = 0;
while ( m < n ){
    // ... something senseless ...
    m++;
    n -= 2;
}
```

The For Loop: Common Errors

Semicolon after the increment:

```java
for (int j = 0; j < 100; j++;) System.out.println(j);
```

Semicolon after closing parenthesis:

```java
for (int k = 0; k < 10; k++){
    System.out.println(k);
}
```
The println is executed only once, after the loop exits (prints 10)

Infinite loop due to careless loop condition:

Example: The following intended to count from low up to high-1.

```java
int low = ...
int high = ...
for (z = low; z != high; z++){
    ...
    // what if high < low?
}
```
Random Number Generation

Pseudo-Random Numbers: Random numbers generated in Java (and all other programming languages) are not truly random. They are simply so unpredictable that they behave as if they are random for all practical purposes.

Seed Value: A sequence of pseudo-random numbers is uniquely determined by an initial seed value, which is given as a long integer.

Math.random( ): Returns a pseudo-random double \( r \) in the range

\[ 0.0 \leq r < 1.0 \]

It works by invoking Java's more general Random number class, which resides in the java.util package.

Java's Class Random

Package: java.util (import java.util.*; )

Constructors: Each call to "new Random" creates a new pseudo-random sequence.

- Random( long seed ): Creates a new sequence using the given seed. Given the same seed, the same sequence is generated every time.

- Random( ): (Default constructor) Creates a sequence using the time of day as the seed. The sequence is different every time you run it.

Getting a new Random Value:

- nextBoolean( ): Returns a random boolean
- nextInt( ): Returns a random int (over the entire range of int's)
- nextInt( int n ): Returns a random int \( r \) over the range \( 0 \leq r \leq n-1 \).
- nextDouble( ): Returns a random double \( r \), where \( 0.0 \leq r < 1.0 \).
- Also: nextFloat( ), nextLong( ), setSeed( long seed )

Why seeds?
The "break" Statement

We saw that the break statement exits from a switch statement. It can also be used to exit immediately from any loop:
- while
- do-while
- for

Example: Generate up to 500 random numbers, but exit the loop as soon as a value less than 0.01 is generated.

```java
int count;
for (count = 1; count <= 500; count++) {
    if (Math.random() < 0.01) break;
}
System.out.println("count = " + count);
```

The "break" Statement

Warning: A break "violates" the loop's natural structure, and can be hard on the reader (particularly if the loop is large). It is best to avoid them, unless it is needed to keep the code simple.

Example: We can easily avoid the break in the above example, by creating a boolean variable.

```java
boolean foundIt = false;
for (count = 1; (count <= 500 && !foundIt); count++) {
    if (Math.random() < 0.01) { foundIt = true; count--; }
}
System.out.println("count = " + count);
```
The “continue” Statement

The `break` statement exits entirely from a loop. The `continue` statement is similar, but jumps immediately to the test portion of the loop, ready to start a new iteration.

Example:

```java
count = 1;
while (count < 20) {
   \(// \text{continue jumps here}\)
   sum += count;
   if (\(...\)) break;
   if (\(...\)) continue;
}
System.out.println("Done"); \(// \text{break jumps here}\)
```

Warning: We only mention `continue` for completeness. `break`: Is usually bad practice. Use it very sparingly. `continue`: Is considered bad practice. Avoid it altogether.