CS206

More Inheritance
Generics

From Last Class
Abstract, Encapsulated, Modular

- Abstract
  - Things/properties shared by all instances of the class. (class should be pencil, not YellowPencil)
- Encapsulated
  - Others cannot "see" what is inside the class. So the inside can change without effect on other classes.
- Modular
  - There are many interacting components.
    - If one fails, it is easier to identify and debug (assuming proper encapsulation).

OOP Design

- Instance variables keep track of the states of an object
  - the only place data is stored
- Methods assume all instance variables are always up-to-date
  - why?
- Each method is responsible for one task and updating the related variables only

Access in Java
Implementation / Enforcement of Encapsulation

- public
  - Usable by every other class
- "" (pronounced package)
  - Usable by every class in the "package"
    - All assignments in this class will use a single package
    - denoted by "package xxx.yyy.zzz" at top of file
    - https://docs.oracle.com/javase/8/docs/api/
    - packages are usually closely aligned with directory structures
- protected
  - Usable by all extenders of the class
- private
  - Usable only within the class

Per Encapsulation, always use most restrictive sensible access

CSStudent and Inheritance

```
public class Student {
   private String name;
   private final String
   id;
}

public class CSStudent extends
Student {
   private boolean isMajor;
}
```

Go to eclipse and work through toString methods

Source Code Organization

- Good Practice: Each project under its own subdirectory
  - directory name = project name
  - Eclipse follows this practice
- One public class per file
- Name of the file must match public class name

Method Overriding

- Inherited methods from the superclass can be redefined/changed
  - "signature" stays the same
    - signature = name+type of all args
  - @Override
    - Not required (it is for this class) but is "best practice" to use
- The appropriate version to call is determined at run time
- `toString` is overridden, twice!

Method Overloading

- Overloading occurs when two methods have the same name but different parameters
- Other languages may not allow overloading.

```
int a(int x) ;
int a(int x, int y);
int a(float y);
int a();
```

```
int a(int x);
int a(int y);
float a(int x);
```

Parsing strings

- Two basic methods
  - Scanner — previously discussed
  - to split on something other than " ", use useDelimiter("delim") method

```
String s="get,thee,to,a,nunnery";
Scanner s2 = new Scanner(line);
s2.useDelimiter(",");
```

- split method of String `string.split(delim)`
  - split a string into an array of Strings based on matching delimiter

```
String s = "neither,a,borrower,nor,a,lender,be";
String[] tokens = s.split(",");
```

Parsing strings
Putting it all together

Create a class with the following methods:

```
/* if true in future, use Scanner, if false, use String.split */
void setUseScanner(boolean uS);

/* prints every word, one word per line in the named file */
void words(String fileName);

/* prints every word after the number in startingWord
     one word per line */
void words(String fileName, int startingWord);

/* print numWords after startingWord, one word per line /*
void words(String fileName, int startingWord, int numWords);
```

In Java, this is referred to as “implementing an interface”

There are lots of reasons to specify an interface, this is one.

Generics

- A way to write classes and methods that can operate on a variety of data types without being locked into specific types at the time of definition
- Write definitions & implementations with "Generic" parameters
- The generics are instantiated (locked down) when objects are created

Generic Methods

```
import java.util.Random;
/************************
 * Author:  G. Towell
 * Created: August 28, 2019
 * Modified: August 29, 2019
 * Purpose:
 *    Generic Methods
 ************************/
public class Genere {
    public static void main(String[] args) {
       Integer[] jj = {1,2,3,4,5,6, 7, 8, 9};  // NOTE AUTOBOXING!!!
       Genere.randomize(jj);
       for (int j : jj)
           System.out.println(j);
       String[] ss = {"A", "B", "c", "d", "E", "F"};
       Genere.randomize(ss);
       for (String s : ss)
           System.out.println(s);
    }

    public static <T> void randomize(T[] data) {
       Random r = new Random();
       for (int i=0; i<data.length; i++) {
           int tgt = r.nextInt(data.length);
           swap(data, tgt, i);
       }
    }

}
```

Write a generic swap method!

Generic Class

```
import java.util.Scanner;

public class Genere2<A> {
    private double amt;
    private A other;
    public Genere2(A other, double amt) {
     this.other = other;
     this.amt=amt;
    }

    public static void main(String[] args)
    {
     Genere2<String> gg = new Genere2<String>("ASDF", 44.5);
     System.out.println(gg);
     Genere2<Double> g3 = new Genere2<Double>(99.5, 44.5);
     System.out.println(g3);
     Genere2<Scanner> g4 = new Genere2<Scanner>(new Scanner("Now is the time for all good")
99.8);
     System.out.println(g4);
    }


}
```

write a toString function for this class

Generics Restrictions

- No instantiation with primitive types
  - `Genre<Double>` ok, but `Genere<double>` is not
- Can not declare static instance variables of a parameterized type
- Can not create arrays of parameterized types
  - but you can create an array of `Object` then cast
    - `(T[])new Object[10]`

Nested Class

- A class defined inside the definition of another class
- When defining a class that is strongly affiliated with another
  - help increase encapsulation and reduce undesired name conflicts.
- Nested classes are a valuable technique when implementing data structures
  - represent a small portion of a larger data structure
  - an auxiliary class that helps navigate a primary data structure
  - **ONLY** place that public instance variables are acceptable
    - They aren't really public