

Notices

Project02 due **tonight**

ALL late work due by **Wednesday night**

- You must have remaining late-day-credits
- You must **email me** about any late/ungraded work!

Last office hours are Wednesday!

- But I will check emails

Java File I/O Revisited

```
File students = new File("/users/phaskell/CS112/roster.txt");

boolean students.canRead();
boolean students.canWrite();
boolean students.isDirectory();
boolean students.delete(); // careful!
boolean students.exists();
int students.length(); // if 'students' is a file
String[] students.list(); // if 'students' is a directory
```

A File just links to a file in the computer filesystem. Can't do any reading/writing with it

Java Stream Types

FileReader, **Scanner**

- `FileReader fr = new FileReader(new File(path));`
- Reads a char or an array of chars from the File
- Handles mapping the computer's native character set to (from) Java's UTF-16
- Scanner...

FileWriter, **PrintWriter**

- FileWriter writes a char or array of chars to a File
- PrintWriter prints standard data types as text
- `print(char)`, `print(double)`, `print(String)`, `println(...)`, etc

(There are other Readers and Writers that handle bytes, objects, etc)

LOOK AT EXCERPT.java and run it!

Code-along

Read keyboard input until it ends

Convert to uppercase

Write to file "output.txt"

Open file and get length in bytes.

See if contains character '\$'

Keyboard input ends if:

- Redirected from a file
- End-of-file character entered



Put Packages
topic here?

See Overflow slides: prez0 there

Enumerations
("enums")

January, February, March, ...

Red, Orange, Yellow, Green, ...

**Biology, Chemistry, Physics,
Geology, ...**

Enumerated Types

Java allows you to define an *enumerated type*, which can then be used to declare variables

An enumerated type declaration lists all possible values for a variable of that type

The values are identifiers of your own choosing

The following declaration creates an enumerated type called `Season`

```
enum Season {winter, spring, summer, fall};
```

Any number of values can be listed

Enumerated Types

Once a type is defined, a variable of that type can be declared:

```
Season time;
```

And it can be assigned a value:

```
time = Season.fall;
```

The values are referenced through the name of the type

Enumerated types are *type-safe* – you cannot assign any value other than those listed

Ordinal Values

Internally, each value of an enumerated type is stored as an integer, called its *ordinal value*

The first value in an enumerated type has an ordinal value of zero, the second one, and so on

However, you cannot assign a numeric value to an enumerated type, even if it corresponds to a valid ordinal value

Enumerated Types

The declaration of an enumerated type is a special type of class, and each variable of that type is an object

The `ordinal` method returns the number value of the object

The `name` method returns the name of the identifier corresponding to the object's value

See `IceCream.java`

```

//*****
//  IceCream.java      Author: Lewis/Loftus
//
//  Demonstrates the use of enumerated types.
//*****

public class IceCream
{
    enum Flavor {vanilla, chocolate, strawberry, fudgeRipple, coffee,
                 rockyRoad, mintChocolateChip, cookieDough}

    //-----
    //  Creates and uses variables of the Flavor type.
    //-----

    public static void main (String[] args)
    {
        Flavor cone1, cone2, cone3;

        cone1 = Flavor.rockyRoad;
        cone2 = Flavor.chocolate;

        System.out.println("cone1 value: " + cone1);
        System.out.println("cone1 ordinal: " + cone1.ordinal());
        System.out.println("cone1 name: " + cone1.name());
    }
}

```

continued

continued

```
System.out.println();
System.out.println("cone2 value: " + cone2);
System.out.println("cone2 ordinal: " + cone2.ordinal());
System.out.println("cone2 name: " + cone2.name());

cone3 = cone1;

System.out.println();
System.out.println("cone3 value: " + cone3);
System.out.println("cone3 ordinal: " + cone3.ordinal());
System.out.println("cone3 name: " + cone3.name());
}
```

continued

```
System.out.println("cone1 value: " + cone1.value());
System.out.println("cone1 ordinal: " + cone1.ordinal());
System.out.println("cone1 name: " + cone1.name());
System.out.println("cone2 value: " + cone2.value());
System.out.println("cone2 ordinal: " + cone2.ordinal());
System.out.println("cone2 name: " + cone2.name());

cone3 = cone1;
System.out.println("cone3 value: " + cone3.value());
System.out.println("cone3 ordinal: " + cone3.ordinal());
System.out.println("cone3 name: " + cone3.name());
System.out.println("cone3 name: " + cone3.name());
}
```

Output

```
cone1 value: rockyRoad
cone1 ordinal: 5
cone1 name: rockyRoad
cone2 value: chocolate
cone2 ordinal: 1
cone2 name: chocolate
cone3 value: rockyRoad
cone3 ordinal: 5
cone3 name: rockyRoad
```

Enumerations

For our class Card, could use an enumeration for the suit

Enumeration for the card values? Maybe, but since there are actual point values tied to the card values, we might want a full class. We want more functionality than we get from an enum.

Probably want enum in its own file, so it can be used by multiple other .java files:

- Card.java, CardDeck.java

Other
"unusual"
syntax



Final classes and methods

A method declared "final" cannot be overridden

```
class Server {  
    final void connectToHardware() { ... }  
}
```

It is not allowed to extend a final class at all

- ...usually for reasons of safety or security

```
final class PasswordStorage {  
    ...  
}
```


Lambda expressions

I don't love 'em...lots of other people do

They started in other computer languages and Java community felt it needed them...

A functional interface is an interface or abstract class with exactly one abstract method

```
interface MathFunction {  
    double mathFcn(double inp);  
}
```

Lambda expressions

A lambda expression is an instance (object) of a functional interface

- It can be used anywhere an object of that interface is needed
- It has a VERY compact definition, without the function name

For example,

```
JButton myButton = new JButton("Press Me");  
myButton.addActionListener( (event) -> {state = !state;} );
```

Syntax:

- Parameters inside parentheses. Can omit parens if only one argument
- ->
- statements in curly braces. Can omit braces and semicolon if only 1 statement

What is the abstract method in this example? actionPerformed()

Lambda expressions

Since a lambda expression is an object, it can be assigned to a variable

```
ActionListener toggle = ev -> state = !state;  
ActionListener turnOn = ev -> state = true;  
ActionListener turnOff = ev -> state = false;  
...  
myButton.addActionListener( turnOff );
```

Look at ShowLambda.java

We talked about PROCEDURAL PROGRAMMING and OOP. Another style is FUNCTIONAL PROGRAMMING,

Writing programs that consist only of functions that take inputs and generate outputs with no "side effects",
i.e. no stored state.

Why? Easier to write bug-free programs. But harder overall to write programs, in my opinion.

One of most popular functional languages is Haskell. I didn't write it, nor any relative of mine. I don't know it.