Csc18a Chapter 10 Notes

**Polymorphism and Interfaces**

Core Topics:

* Polymorphism
* Abstract classes and methods
* Interfaces

Notes:

**API**

* Application programming interface

**Refactor tool**

* Netbeans tool
* Rename
  + Renames class name
    - Auto changes every instance of the name
* Encapsulate Fields
  + Contains a shortcut method for creating getters and setters
    - Auto Generates code
  + Location:
    - Toolbar/Refactor/encapsulate fields…
    - Checkmark methods to create
    - Refactor

**Polymorphism**

* Allows you to program “in general” instead of “specifics”
* Enables you to write programs that process objects that share the same super class as if they were all objects of the superclass.
  + Basically allows you to create a superclass of a category that acts differently within each subclass
  + Ie. Enemy super class with ghost, goblin, skeleton subclasses.
* A program implementing polymorphism is referred to as Layered Software Systems

|  |  |
| --- | --- |
| Differences | |
| Interface | Abstract |
| * No Variables * Implementations allowed * Abstract methods | * Variables * No implementations * Abstract methods * Can have non-abstract methods that are implemented |

**Interface**

* Teacher def.
  + Contract of public member functions that tells us what functions a program needs to work.
  + The implementation of our functions is up to us.
  + \*\*\* **Member function declarations only!**
    - No member variables allowed
    - Must be public or protected
* Purpose:
  + Defines ad standardizes the how objects interact with each other
    - Creates an object that requires certain functionalities
  + Useful for assigning common functionality to possibly unrelated classes.
    - Referred to as disparate classes
    - Ex) payable interface which receives pay amount and an employee and invoice class that receives and returns it.
  + Allows objects to be processed polymorphically
* Uses **interface** keyword
  + Also located under: right-click project/new/Java Interface
* Contains *only* constants and abstract methods
* **Syntax:**
  + public **interface** geometry {

public double area(); //Calculates area

public double perimeter(); //Calculates perimeter

public double surface\_area(); //Calculates surface area

public double volume(); //Calculates volume

}

**Using an interface**

* Uses **implements** keyword
* Creates an object with the interface
* **Syntax:**
  + Public class Triangle **implements** geometry{

//code…

}

* A class that doesn’t use all interface methods is an abstract class and *must be declared abstract*.
  + Not doing so results in an error
* An interface is often used in place of an abstract class when there is no default implementation
  + No fields, no default method implementations
* List of available interfaces on pg. 423 of Java text book

**Abstract classes**

* Similar to templates in C++
* Teacher def.
  + A blue print/template for an object
  + Allowed to have member variables of any kind
    - Member functions and abstraction member functions.
  + Without the abstract member functions, it’s just a normal class
  + \*\*\***Cannot be instantiated**
* Purpose
  + To provide an appropriate superclass that other classes can inherit and share a common design.
  + Meant to be overridden.
* Can’t…
  + Provide implementations
  + Be private
* **Syntax:**
  + Public **abstract** void fName();
    - Uses **abstract** keyword
    - Must be explicitly declared abstract even if that class contains some non-abstract methods.
* Constructors and static methods cannot be abstract
  + Constructors are not inherited, thus cannot be implemented in abstracts

**Abstract member functions (methods)**

* Teacher def.
  + Functions that are declared but NOT implemented.
* Uses
  + An abstract method can be implemented without any code in order to specify that all child classes must have it.
    - Ex) public employee {

//code…

Public **abstract** double earnings();

}

**Abstract subclasses**

* Referred to as Concrete classes
* Implemented by inheritance
  + Ex) public class SalariedEmployee **extends** Employee{

//code…

}

* Requires @Override keyword to use required abstract methods
  + If abstract method is not overridden, it will inherit super class’s method.

**Polymorphic Processing**

* Non-Polymorphic Processing
  + Done by creating variables for each concrete item
    - Ex)
    - SalariedEmployee employee1 = new SalariedEmployee(//Arguments);

HourlyEmployee employee2 = new HourlyEmployee (//Arguments);

CommisionEmployee employee3 = new CommisionEmployee (//Arguments);

* Polymorphic Processing
  + Done by process each object within the same algorithms or with the same method call
  + Array processing:
    - Ex)
    - Employee[] employees = new Employee[3];

Employees[0] = employee1;

Employees[1] = employee2;

Employees[2] = employee3;

* + - * This is valid since all objects are of the Employee class.
  + Method call:
    - Ex) for(Employee currentEmployee : employees){

//Code…

System.out.printf(“Earned $%, .2f%n%n”, **currentEmployee.earnings()**);

}

* + - * Method call **.earnings()** is implemented in all concrete classes but acts differently due to their individual overridden methods.
* **instanceOf** operator
  + Determines whether a particular item is part of the super class.
  + Returns Boolean
    - True: If object is referenced
    - False: If object is not referenced

**Downcasting**

* Allows user to change data type of super class to subclass
  + Ex) CommisionEmployee employee = (ComissionEmployee) currentEmployee;
* Helps us avoid certain errors
  + If we needed to assign a subclass to superclass
    - Prevents “incompatible type” error
  + If we need to use methods of subclass in superclass
    - Prevents “cannot find symbol” error
    - However, you cannot call a subclass variable with a superclass method.
      * Superclass methods can only access superclass variables

**Getting the class name**

* Every object knows its own class
* Done with **getClass** method
  + Inherited from object class
  + Returns an object type Class, which contains info about the object’s type, ie. Class name.
* **Syntax:**
  + employees[1]**.getClass()**.getName();

**Classes within classes**

* Referred to as **Inner Classes**

**Final Methods and classes**

* Final methods cannot be overridden
  + This guarantees that the method will be used by all direct and indirect subclasses
  + Private and Static methods are implicitly final, since they cannot be overridden
  + Declaration cannot be changed
  + AKA. Static Binding
* Final classes
  + Cannot be superclass
  + Prevents programmers from creating subclasses for security reasons
* Make what can be final, final.
  + Optimizes code through compilers
* For more insight, visit:
  + <http://docs.oracle.com/javase/tutorial/java/IandI/final.html>

**Note on method calling**

* Do not call overridden methods in constructors
  + Can lead to methods being initiated before the actual class
* Calling static methods in constructors is fine
* Also acceptable with final methods, provided it doesn’t call any overridden methods

**Anonymous Object**

* A temporary object
* Declared by calling the import within a variable
* **Syntax:**
  + Int v = new **java.util.Scanner(System.in).nextInt();**
    - Variable receives next int.