CMPE212 – Reminders

• Assignment 3 due Friday.

• Quiz 2 next week in the lab. Topic Coverage on Next 3 Slides:

Quiz 2 Topics

• Everything up to and including Monday’s lecture, next week.

• Exercises 1 to 8 (but not exercise 2), and assignments 1 and 2 are fair game and would be good preparation for the Quiz. Sample solution for assn 2 will be posted the night before the quiz.

• More details on next two slides:

Quiz 2 Topics

• Quiz 1 Java topics, but not Java History or Background (“How Java Works”).

• Method Overloading.

• Catching, Building and Throwing Exceptions.

• Aliasing Objects, Passing by Reference.

• Objects in general. Instantiation.

• Encapsulation – private attributes, constructors, accessors, mutators and other standard methods.

• File I/O from Exercise 4 – concepts only – you won’t have to write file I/O code.

• JUnit Testing – concepts only – you don’t need to write testing code.

• TDD.

Quiz 2 Topics, Cont.

• Packages & Java Modules – concepts only.

• Enumerated Types.

• Inner, Anonymous and Abstract Classes.

• You will be given a list of String methods if you need them.

• One or two coding problems.

• Otherwise, same rules as Quiz 1.

Today

• Inner or “Nested” Classes.

• Anonymous Classes.

• Abstract Classes.

• Start Interfaces. (if we have time)

Inner Classes

• Simply defined as a class defined within a class:

```java
public class OuterClass {
    private class InnerClass {
        Attributes of innerclass
        Methods of innerclass
    } // end InnerClass

    Attributes of OuterClass
    Methods of OuterClass
}
```
Inner Classes, Scope Rules

- See the demo program InnerClassDemo.java, used with TestInnerClassDemo.java.
- The inner class can easily get at all the attributes and methods of the outer class.
- The outer class can also access the attributes and methods of the private inner class, after it instantiates the inner class.
- However, all private classes, methods and attributes are hidden outside the outer class.

Inner Classes, Why private?

- If you are declaring a non-static object inside your class, you are only doing so because you want to hide it away.
- This object can only be used inside the outer class and is hidden everywhere else.
- How else can you make a class private and still be able to use it?

Inner Classes, Cont.

- So, what's the point of private inner classes?
- One reason - When you wish to use a small class (in terms of the size of its definition) but don't want to bother creating a separate class definition file.
- This is often used with Linked List definitions to define the node object, for example.
- Yes, you can have an inner class inside an inner class!

Aside - Using Reflection with Inner Classes

- See the demo program InnerClassDemo.java, used with TestInnerClassDemoV2.java.
- There is a way to invoke the private method in the private inner class from outside the class using Reflection.
- (You can prevent this type of misbehaviour through the module descriptor or by using security manager objects. I won't get into this...)

Using Reflection with Inner Classes, Cont.

- Reflection is a set of techniques that allow you to discover and use any & all public/private/protected members of class through a Class<T> object. Even if all you have is the byte code file!
- The demo also uses some advanced stuff: wildcarded generic types, the .class attribute, Method, Constructor and Class objects, as well as "multi-catch" try/catch blocks.
- Was this easy to write? NO!!
- Should you do this stuff? Maybe not...

public static Inner Classes

- Why declare an inner class public static?
- It would allow you to "categorize" methods into topical groups. Invoke them as below:
  - First:
    - TestClass tc = new TestClass();
  - Then, invoking static methods from public static inner classes:
    - tc.Group1.method();
    - tc.Group2.anotherMethod();
Anonymous Classes, Example

```java
public class AnonymousClassDemo {
    public static void main(String[] args) {
        MessageSender ms = new MessageSender() {
            public void sendGreeting(String name) {
                System.out.println("Hello " + name + "!");
            }
        };
        ms.sendGreeting("Alan");
    }
}
```

Anonymous Classes, Example - Cont.

- Displays "Hello Alan!" to the console (Wow!)
- MessageSender is an interface, not an Object:

```java
public interface MessageSender {
    void sendGreeting(String name);
}
```

- So, you have not actually named the Object that contains the definition of the method sendGreeting()!
- ms is an Anonymous Object.

Anonymous Classes, Cont.

- Some coders will tell you that anonymous classes are just classes written by lazy coders.

- This is true, but you will see these little beasts used with GUI coding... (especially by lazy GUI coders, or the GUI wizards in NetBeans and Eclipse).

- Java 8 (and newer) now has a very tidy solution to using messy anonymous classes in a GUI program – by using Lambda Functions instead – more on these later!

Anonymous Classes, Example - Cont.

- How would you do this in a "non-anonymous" way?

- First, create a named class that implements the interface:

```java
public class ImplementingSender implements MessageSender {
    public void sendGreeting(String name) {
        System.out.println("Hello " + name + "!");
    }
}
```

- Now, iAmNamed is an instance of a named type.

Anonymous Classes, Example - Cont.

- Then, instantiate this class and invoke the method:

```java
public class TestSender {
    public static void main(String[] args) {
        ImplementingSender iAmNamed = new ImplementingSender();
        iAmNamed.sendGreeting("Alan");
    }
}
```

Next...

- Interfaces are completely abstract – they cannot contain much in the way of concrete code.

- But first, let’s look at something in-between an interface and a fully concrete class – an abstract class:
abstract Classes

- It is not unusual to declare a class in the root of a hierarchy to be abstract:
  ```java
  public abstract class MyClass ...
  ```
- Any class declared this way cannot be instantiated.
- It can only be extended.
- Unlike an interface, an abstract class can also contain concrete method definitions and any kind of attribute.
- If a class has one or more abstract methods, the class must be declared abstract as well.

Aside – Preventing Instantiation, Cont.

- Before: write a private default constructor.
- But, you can also name any class as abstract, even if it does not have any abstract methods. This will also make sure that this class cannot be instantiated.
- The advantage here is that you can now have public constructors that can be used by child classes. A non-abstract child class can be instantiated.

abstract Classes, Cont.

- abstract methods have no code in them. For example:
  ```java
  public abstract String getListing();
  ```
- A class that extends an abstract class must override all the abstract methods in the class, unless it wants to be abstract too.
- “Overriding” means that you must have a concrete implementation of that method in the child class.
- Unlike an interface, you need to write public abstract for each abstract method signature.

abstract Classes, Cont.

- Why bother?
- An abstract class forces sub-classes to define certain methods. This helps ensure that the hierarchy has a consistent design.
- Also, when declaring a method in a very abstract class, then you don’t have to worry about what to do in the method body, especially if it must return a value.
- One way to provide the mechanism for polymorphism!

Aside – Polymorphism (A Quick Peek)

- We will learn more about this important aspect of OOP later, but for now:
- Polymorphism is when a pointer of a parent class type ends up pointing to different child class objects at runtime. Also called Dynamic Binding. The process must also satisfy early binding:
- Early Binding is satisfied when the parent class also owns the method that will end up being invoked from the morphed child class objects. The use of interfaces and abstract classes can make for easier coding to satisfy early binding.

Interfaces in Java 7 and Older

- Interfaces used to just contain:
  - constant attributes and/or
  - abstract methods.
- Abstract methods consist of just the method header – there is a semi-colon instead of {}.
- (Abstract classes can contain both concrete and abstract methods, along with any kind of attribute.)
Interfaces in Java 8

• In addition to the above, we can also have:
  – Default methods.
  – Static methods.

Java 8 Interface Default Methods

• Must have the keyword default at the beginning of the method header.
• These methods will not be empty, but can contain implementation code.
• This will be a “default” implementation for this method.
• This method can be overridden or inherited by a class implementing the interface, or by another interface extending this interface.
• (Yes, you can have an interface hierarchy!)

Java 8 Interface Default Methods, Cont.

• The purpose is to reduce the work required when methods are added to interfaces in existing hierarchies.
• You do not *have* to implement the default method.
• (This also makes interfaces more like abstract classes…)

Java 8 Interface static Methods

• They behave just the same way a static method behaves in a normal class.
• A static method is not inherited.
• Also simplifies design.

Interfaces in Java 9

• In addition to the options listed above, Java 9 interfaces can have private static and normal private methods, just like those in a normal class.
• The purpose is to simplify the construction of the interface by providing a means to eliminate common code in other methods.

Interfaces in Java 9, Cont.

• To summarize, an interface can now contain:
  – public static final attributes.
  – abstract methods.
  – default methods.
  – static methods.
  – private methods.
  – private static methods.
• The keywords public and abstract are still not required.
• What is not allowed?
Interfaces in Java 9, Cont.

- So, now an interface does not have to be just a design specification.
- It can contribute to and simplify the construction of an object hierarchy.

Interfaces, Cont.

- Interfaces do not extend `Object`.
- Interfaces can extend multiple interfaces (but not other classes).
- Classes can implement one or many interfaces:

  ```java
  public class Test implements interface1, interface2, interface3, ... {} 
  ```
- If two interfaces have the same method name, the implementing class must implement that method, even if it is a `default` method.

Interfaces, Cont.

- An interface cannot be instantiated, but you can use the interface type as if it is a class.
- The interface acts as a "stand-in" for the concrete object that will replace it when the program is running.
- The interface "guarantees" the required behaviour of the object it is standing in for.

Interfaces, Cont.

- Attributes in interfaces can only be `public final static`, you cannot use `private`. You don't even have to specify `public final static` as it is assumed. Constants must be initialized.
- In Java 8 and 9, you should not and sometimes cannot use the `public` and `abstract` keywords in method headers. They are not needed anyways...

Interfaces, Cont.

- A class that implements an interface must have a concrete implementation of every abstract method signature in the interface.
- The class can just accept (inherit) default methods, can use `static` methods and will not see private methods.
- When designing an interface, you must try to use unique method and constant names, in case your interface becomes part of a multiple implementation.