CMPE212 – Reminders

• Assn 4 not posted yet.

• Quiz 2 this week in the lab. Topic Coverage in last Wednesday’s lecture. This lecture’s material is fair game.

Today

• Polymorphism Demo using the DoRunRun interface.

• Inheritance.

Demonstrating Polymorphism and Safe Casts

• Let’s first create a bunch of classes that implement the DoRunRun interface.

• Create a collection of instances of these classes.

• Feed the collection to two methods:
  – One that accepts a DoRunRun[] array, and
  – One that accepts an Object[] array.
  – (Ignore the generic version, for now.)

• What’s the difference in how the two methods operate?

Notable:

• See how DoRunRun can be used as a type.

• The DoRunRun[] array type can hold instances of any class that implements DoRunRun.

• The Object[] array type can hold any Object...

• If the method accepts DoRunRun[], then each element can be of the type DoRunRun, and we know that each element will have a concrete version of the run() method.

Noteable, Cont.:

• As you iterate through the collection, the variable “one” morphs into its actual object at run-time, so you know the correct run() method will be invoked.

• If the argument to the method is of type Object[], the elements in the collection are also of type Object.

• An Object does not have a run() method. Early binding will fail if you try to invoke run() from a variable of type Object.

Noteable, Cont.:

• So, first you have to cast the variable to a type that does have a run() method.

• What happens if the Object[] array has something in it that does not implement DoRunRun?

• Which method version is more prone to failure?

• We will see (soon!) that the mechanism for Polymorphism can be achieved through inheritance as well, not just through implementation.
Hierarchy Example from the Java API

- A quick look at part of an existing hierarchy – the Collection<T> hierarchy – it won’t hurt and it shows the use of Interfaces and Abstract classes:

A “Tree” View from the API Docs:

Inheritance

- An OOP design technique that allows you to add to, modify, replace or simply adopt the methods and attributes in an existing class.
- The class that you are modifying is called the parent or super class, and the resulting class is called the child or sub class.
- The child or sub-class inherits all the public attributes and methods of the parent or superclass.
- The child class extends the parent class.
- The sub-class is always more concrete or less abstract than the super class.

Why Bother?

- One reason is code re-use.
- It is also a design tool that allows the coder to more easily model a real-life structure.
- You can also use inheritance to enforce code structure on child objects (using interfaces and abstract classes).
- And it can be part of the polymorphism mechanism.
- Many classes in the Java API are intended for use as parent classes.

Real Life Hierarchies

- Can you think of any?
  - Flora, Fauna
  - Rocks & Minerals
  - Food Menus
  - Fasteners
  - Vehicles
  - Computers
  - etc!
Music Hierarchy

- For example, see:
  

- Almost everything must belong to a hierarchy.
- We often need to model real-world hierarchies in code. Easier to do because we already have the model.

Example

- Our classroom:

  ![Classroom Diagram]

- Both Professor and Student are sub-classes (or children) of the super-class or (parent) Person.

Example, Cont.

- Note that:
  
  - Professor "is a" Person.
  - Student "is a" Person.
  - Professor "is not a" Student.
  - Student "is not a" Professor.

Some Goals of Hierarchy Design

- All attributes in an instance (its "state") should be defined with meaningful values.
- Minimize code repetition.
- The same attribute should not be declared in more than one class.
- A child class should be more concrete than its parent class.
- Design for polymorphism:
  - Use abstraction (abstract classes and interfaces) to ensure common behaviour and to allow for polymorphism.

Some Goals of Hierarchy Design, Cont.

- Control and minimize the public interface of all classes.
- Try not to have any "pass through" classes that do not contribute anything.
- Make sure the structure is extensible.
- Use a consistent variable and method naming scheme throughout the hierarchy.
- Follow the "real world" hierarchy as closely as possible.
In Java, every new object you define is automatically a child of the Object class. So our actual structure is:

- Object
- Person
- Professor
- Student

This is a simplified UML Class Diagram. The upward pointing arrow shows the “is a” relationship.

So:
- Person is a Object
- Professor is a Person
- Student is a Person
- Professor is a Object
- Student is a Object

Object is the “Universal” parent class for every object in Java – in your code, in the API, everything!

So, we are inheriting all the public members of this class – whether we want this stuff or not.

What members are we getting and how do we find out?

Example, Cont.

Suppose we wish to store the description of every person in this class (me included!) in one (small) array.

What array declaration would you use?

The following example code is greatly simplified. It does not have any error checking, for example.

```java
public class Person {
    private int age;
    private String name;
    private String gender;

    public Person (int a, String n, String g) {
        age = a;
        name = n;
        gender = g;
    }
}
```
public class Professor extends Person {
    private int numExamsToCreate;
    public Professor (int a, String n, String g, int e) {
        super(a, n, g);
        numExamsToCreate = e;
    } // end Professor constructor
} // end Professor class

public class Student extends Person {
    private int numExamsToWrite;
    public Student (int a, String n, String g, int e) {
        super(a, n, g);
        numExamsToWrite = e;
    } // end Student constructor
} // end Student class

super Keyword
- Provides a reference to the immediate parent class.
- In the examples above, it was used to call the constructor in the parent class, Person.
- Note that the compiler will force you to invoke super in a child class' constructor, and it must be the first line in the constructor.

public class ClassroomDB {
    public static void main (String[] args) {
        Person[] db = new Person[20];
        db[0] = new Student(18, "Fred", "male", 7);
        db[1] = new Student(19, "Melissa", "female", 7);
        db[2] = new Professor(29, "Alan", "male", 1);
    } // end main
} // end ClassroomDB

Polymorphism, Again
- Pointers of type Person are pointing to objects of type Student and Professor at runtime.
- At runtime the pointers will morph into their actual object types – this is late binding or dynamic binding.