More About Inheritance

• Inheritance provides the power of object-oriented languages, but it is also a complex mechanism
  – Static type checking is provided at compile type
  – But most objects are created at runtime, so we must distinguish between static and dynamic types
  – Methods are looked up dynamically and we need to understand this mechanism

• Other topics we will cover are
  – Method overriding and method polymorphism
  – A new scoping level called protected
  – Using Interfaces to simulate multiple inheritance
A Review of DoME and Modifications

• We reviewed the second version of DoME based on inheritance and noted some weaknesses

• A primary weakness is that the details that are unique to CDs or Videos are not printed

• As a first attempt to fix this problem, we examine the third version of DoME

• This version introduces a print method in the CD and Video subclass to print the unique features of these subclasses

• Did this fix the problem? If not, what can be done?
How can we fix this problem?

• Here is our predicament
  – If the print is in Item, we only get the shared information
  – If there is also a print in the subclasses, it is executed and the general information is not printed

• Discuss some solutions to solve this problem so that both sets of information are printed
Some Methods Common to All Objects

• A list of methods
  – clone – makes a copy of an object
  – equals – checks whether two instances are equal
  – hashcode – defined for any object for use with collections such as HashMap
  – toString – how should information about an object be presented in String format
  – Other methods, such as notify and wait, are beyond the scope of this course

• Which methods would you want to override and when would this occur
Using toString

• It is common to override toString
• How would this be done for Item, CD, and Video?
• Which method in Item, CD, and Video could benefit from overriding toString
• Make these changes now
Formalizing These Concepts

• The types of a variable
  – The static type of a variable is how it was declared in the source code
  – The dynamic type of a variable is the type of the object that is currently stored in the variable
  – Because of inheritance, these types may differ

• An example from DoME
  – The static declaration of a variable may be of type Item
  – When the object is actually created at runtime, the type might be CD
  – Since a CD is an Item based on inheritance, this runtime creation of the object is permitted
Dynamic Method Lookup

• We have already learned that a method that is declared at a higher level of an inheritance hierarchy can be overridden at a lower level.

• When a method is called on an instance of an object at runtime, the method is found as follows:
  – The method is fetched from the dynamic type of the variable, if it is defined at that level.
  – If the method is not found, the search goes to the parent class, the grandparent class, and so forth until the method is found; the first matching method is used.
  – In DoME, when we added print to the CD, it was the first matching method for print, so the parent method was ignored; this was fixed by having the lower level method call the parent method as well as execute its own code.
Method Polymorphism

• Methods in Java are polymorphic
  – The same method call may exist at different levels of the inheritance hierarchy and behave differently when called
  – The actual method called depends on the dynamic type of the variable at the time of call

• An example, suppose m1 is declared type Item
  – Suppose CD has overridden the print method in Item but Video has not overridden the print method
  – If m1 currently has a dynamic type of CD and print is called, the print in CD is invoked
  – If m1 currently has a dynamic type of Video and print is called, the print in Item is invoked
  – This behavior can change at runtime based on the dynamic type assigned to m1
Protected Access

• So far we have seen two levels of scoping: public and private
• With inheritance, we introduce a third level, protected, that is intermediate between public and private
• A protected scope means only the class and any of its children can access the field or method
• A comparison
  – Public is accessible to anyone
  – Private is accessible to no one except the class itself
  – Protected is accessible to the class itself and any object lower in the inheritance hierarchy but no one else
Multiple Inheritance

• Java only supports single inheritance, meaning every class can have at most one parent class

• C++ supports multiple inheritance meaning a single class might have multiple parents
  – Multiple inheritance is much more difficult to implement and understand
  – Suppose a class has two parents, if a method is called that is not defined in the local class, a search for that method proceeds up both parent structures
  – Suppose two different methods are found, which one is used? The semantics of multiple inheritance is challenging
Using Interfaces

• Recall that an interface is like a class, however none of the methods have method bodies

• The syntax of a class using one or more interfaces
  
  public class <classname> implements <list of interfaces> { …

• One class can implement multiple interfaces but the class must fully define all the methods in all of the interfaces

• Interfaces and inheritance can be combined, as in
  
  public class <name> extends <parent> implements <interfaces> { …

• As seen in the following example, we can use multiple interfaces to allow Java to have the same advantages as multiple inheritance
A Student Employee - 1

- Consider

```java
public class Person {
    public String getName() {
        // ...
    }
    protected String name;
}

public class Student extends Person {
    public float getGPA() {
        // calculate GPA
    }
    protected float gpa;
    // ... other methods and fields
}

public class Employee extends Person {
    public float getSalary() {
        // calculate salary
    }
    protected float salary;
    // ... other methods and fields
}

// the following is illegal in Java!
// multiple inheritance of classes
public class StudentEmployee extends Student, Employee {
    // implementation of both getGPA() and getSalary() is inherited
    // ... other methods and fields
}
```
A Student Employee - 2

- Here is a picture of our solution

- The dotted lines represent Interfaces

- So how are these relations implemented in Java code?
A Student Employee - 3

• First we define the two interfaces

```java
interface Student {
    public float getGPA();
}

interface Employee {
    public float getSalary();
}
```

• We delegate implementation to other classes

```java
public class StudentImpl implements Student {
    public float getGPA() {
        // calculate GPA
    }
    protected float gpa;
}

public class EmployeeImpl implements Employee {
    public float getSalary() {
        // calculate salary
    }
    protected float salary;
}
```
• We define classes that don’t use multiple inheritance

```java
public class FulltimeStudent extends StudentImpl {
    // method getGPA() and field gpa are inherited
    // .. other methods and fields
}

public class FulltimeEmployee extends EmployeeImpl {
    // method getSalary() and field salary are inherited
    // .. other methods and fields
}

public class StudentEmployee implements Student, Employee {
    public StudentEmployee() {
        studentImpl = new StudentImpl();
        employeeImpl = new EmployeeImpl();
        // ...
    }

    public float getGPA() {
        return studentImpl.getGPA(); // delegation
    }

    public float getSalary() {
        return employeeImpl.getSalary(); // delegation
    }

    protected StudentImpl studentImpl;
    protected EmployeeImpl employeeImpl;

    // ... other methods and fields
}
```

• We define the student employee class
Lab 18: Extending DoME

• This lab has three parts
  – Modify CD and Video so that the toString method is overridden and used to produce the proper printouts
  – Add a new subclass to Item called VideoGame
  – Set up an automated testing program to show that this latest version of DoME works as intended

• A detailed description of this lab appears in the ReadMe file for the lab