# Polymorphism

Han-fen Hu

Tony Gaddis (2019) Starting Out with Java: From Control Structures through Data Structures, 4th Edition

UNLV

# Outline

- Implementation of Polymorphism
- Abstract Classes and Methods
- Usage of Interfaces

UNLV

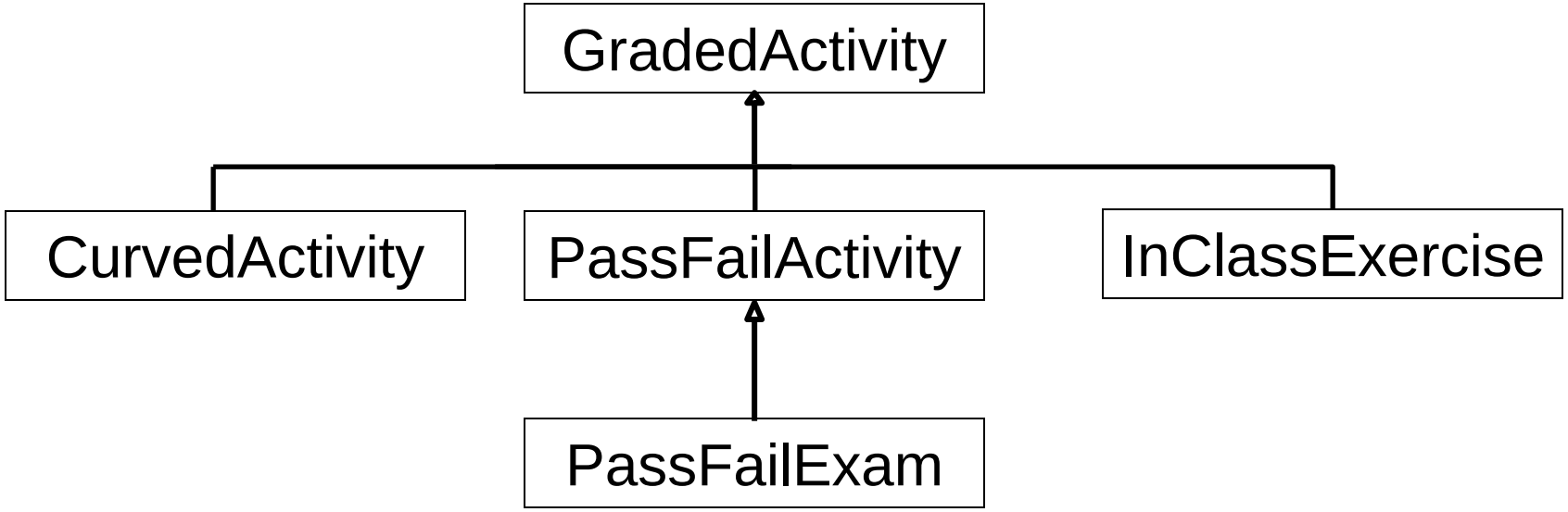
# Polymorphism (1)

- The term *polymorphism* means the ability to take many forms.
- Polymorphism allows a single variable to refer to objects from different subclasses in the same inheritance hierarchy
- For example, if **`Cat`** and **`Dog`** are subclasses of **`Pet`**, then the following statements are valid:

```
Pet myPet;
myPet = new Dog();
. . .
myPet = new Cat();
```

UNLV

# Example of an Inherence Hierarchy



# Polymorphism (2)

- The following statement creates a **`CurvedActivity`** object and stores the object's address in the **`exam`** variable.

```
GradedActivity exam;
exam = new GradedActivity();
exam = new CurvedActivity(1.1);
```

- The **`GradedActivity`** class is the superclass for the **`CurvedActivity`** class.
  - An object of the **`CurvedActivity`** class ***is a*** **`GradedActivity`** object.
  - A **`GradedActivity`** variable can be used to reference a **`FinalExam`** object.
- The reference variable exam is polymorphic

# Polymorphism (3)

- Other legal polymorphic references:

```
GradedActivity exam1 = new InClassExercise("Ex3","2021-03-
  15");
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
```

- The **`GradedActivity`** class has three methods: **`setScore`**, **`getScore`**, and **`getGrade`**.
  - A **`GradedActivity`** variable can be used to call only those three methods.

```
GradedActivity exam = new PassFailExam(100, 10, 70);
System.out.println(exam.getScore()); // This works.
System.out.println(exam.getGrade()); // This works.
System.out.println(exam.getDueDate()); // ERROR!
```

# Lab (1)

- PolymorphicExam.java
  - The program contains an array that represents three exams

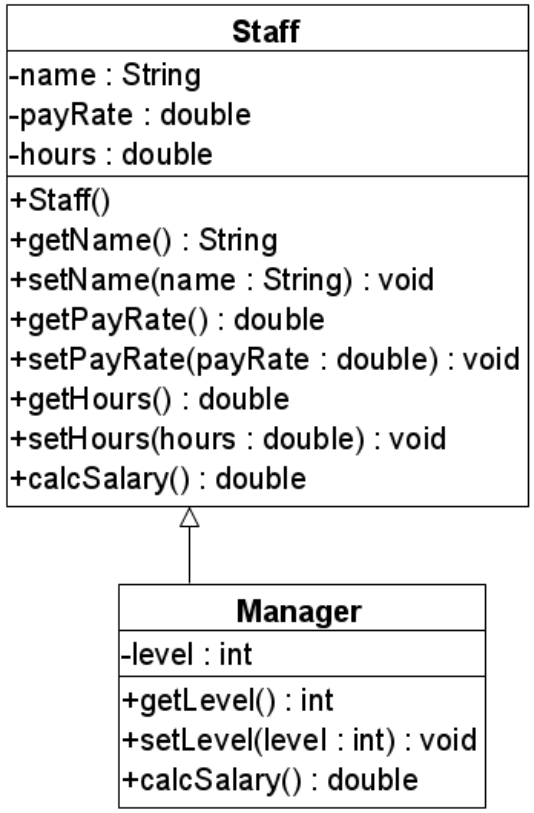
# Dynamic Binding

- If the object of the subclass has overridden a method in the superclass:
  - If the variable makes a call to that method the subclass's version of the method will be run.

```
GradedActivity exam = new PassFailActivity(60);
exam.setScore(70);
System.out.println(exam.getGrade());
```

- Java performs dynamic binding when a variable is polymorphic
  - The Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references.

# Lab (2)




- Payroll.java
  - The application summarize the payroll by adding the Staff or Manager object to an ArrayList.
  - When traversing the ArrayList to print the salary, the program dynamically determine which calcSalary() method to call



UNLV

# Abstract Class

- An *Abstract class* serves as a superclass for other classes.
  - An abstract class cannot be instantiated, but other classes are derived from it.
  - The abstract class represents the generic form of all the classes that are derived from it.
- A class becomes abstract when you place the abstract key word in the class definition.

```
public abstract class ClassName
```

# Abstract Method

- When a class contains an abstract method, it becomes an abstract class
- An *abstract method* is a method that appears in a superclass, but expects to be overridden in a subclass.
  - An abstract method has only a header and no body.
  - An abstract method must be overridden in a subclass.

# Abstract Class and Method: Example

```
public abstract class AbstractClassExample{
    protected int x;
    public void abstract print();
}
```

# Define Abstract Method

- For the abstract method
  - key word **`abstract`** appears in the header, and that the header ends with a semicolon.

```
public abstract void setValue(int value);
```

- For the subclass
  - If a subclass fails to override an abstract method, a compiler error will result.
- Abstract methods are used to ensure that a subclass implements the method

# Lab (3)

- Student.java
  - Abstract class which is generic
- BusinessStudent.java
  - Implement the abstract method and define how a business student would be considered as completing the degree
- BusinessStudentDemo.java
  - Application class to test

# Interfaces (1)

- An *interface* is similar to an abstract class that has all abstract methods.
  - It cannot be instantiated, and
  - All of the methods in an interface must be written elsewhere.
- The purpose of an interface is to specify behavior for other classes.
- An interface looks similar to a class, except:
  - Keyword **`interface`** is used instead of the keyword **`class`**, and
  - Methods that are specified in an interface have no bodies, only headers that are terminated by semicolons.

# Interfaces (2)

- The general format of an interface definition:

```
public interface InterfaceName {
   (Method headers...)
}
```

- All methods specified by an interface are public by default.

# Interfaces (3)

- A class can implement one or more interfaces.
- If a class implements an interface, it uses the **`implements`** keyword in the class header.

```
public class FinalExam extends
  GradedActivity implements Relatable
```

# Fields in Interfaces

- An interface can contain field declarations:
  - All fields in an interface are treated as final.
- Because they automatically become final, you must provide an initialization value.

```
public interface Doable {
   int FIELD1 = 1, FIELD2 = 2;
   (Method headers...)
}
```

  - In this interface, **`FIELD1`** and **`FIELD2`** are final **`int`** variables.
  - Any class that implements this interface has access to these variables.

# Lab (4)

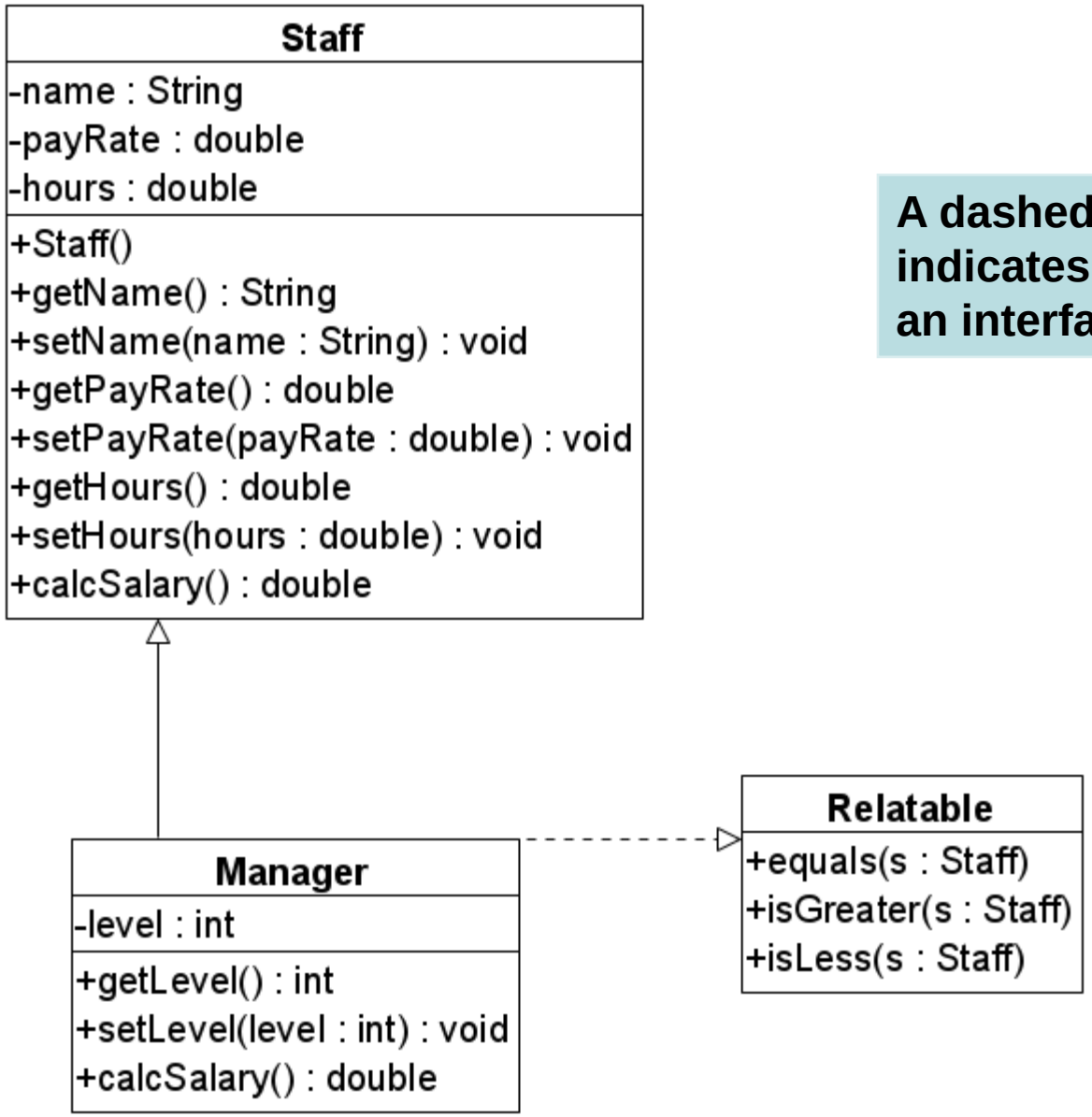
- Relatable.java
  - The interface specify the comparison methods needed
- Manager.java
  - Model class that implements the interface
- ManagerComparisonDemo.java
  - Application class that is used to test the newly implemented methods

# Implementing Multiple Interfaces

- A class can be derived from only one superclass.
- Java allows a class to implement multiple interfaces.
  - When a class implements multiple interfaces, it must provide the methods specified by all of them.
- To specify multiple interfaces in a class definition, list the names of the interfaces, separated by commas

```
public class MyClass
        implements Interface1,
                   Interface2,
                   Interface3
```

# Interfaces in Class Diagram




**A dashed line with an arrow indicates implementation of an interface.**

UNLV

# Polymorphism with Interfaces (1)

- Java allows you to create reference variables of an interface type.
  - An interface reference variable can reference any object that implements that interface, regardless of its class type
- This is another example of polymorphism.

# Lab (5)

- RetailItem.java
  - Interface
- CompactDisc and SreamingMovie
  - Model classes that implements RetailItem
- PolymorphicInterfaceDemo.java
  - Application class to test the classes

# Polymorphism with Interfaces (2)

- A reference to an interface can point to any class that implements that interface.
- You cannot create an instance of an interface.

```
RetailItem item = new RetailItem(); // ERROR!
```

- When an interface variable references an object:
  - Only the methods declared in the interface are available