Inheritance, Polymorphism, and Interfaces

Chapter 8

*JAVA: An Introduction to Problem Solving & Programming, 7*th Ed. By Walter Savitch
ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Inheritance Basics (ch.8 idea)

- Inheritance allows programmer to define a general *superclass* with certain properties (methods, fields/member variables)
- Later, typically, you define a more specific *subclass*
  - Adds new details – the new subclass is a *specialization* of the general *superclass* (can also reverse and *generalize* existing classes…)
- Subclass instances are superclass instances, generally with more information/methods as well (called an *is-a* relationship)
- Subclass inherits all fields of the general superclass. It **extends** (keyword for inheritance) the superclass, often adding new fields and/or methods.
- *Polymorphism*: Can also *override* methods in the superclass, defining a new implementation for subclass methods that were defined. A superclass *variable* referencing a subclass *instance* will use subclass definition.

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

(A) Recipe for Using Inheritance (ch.8 outline)

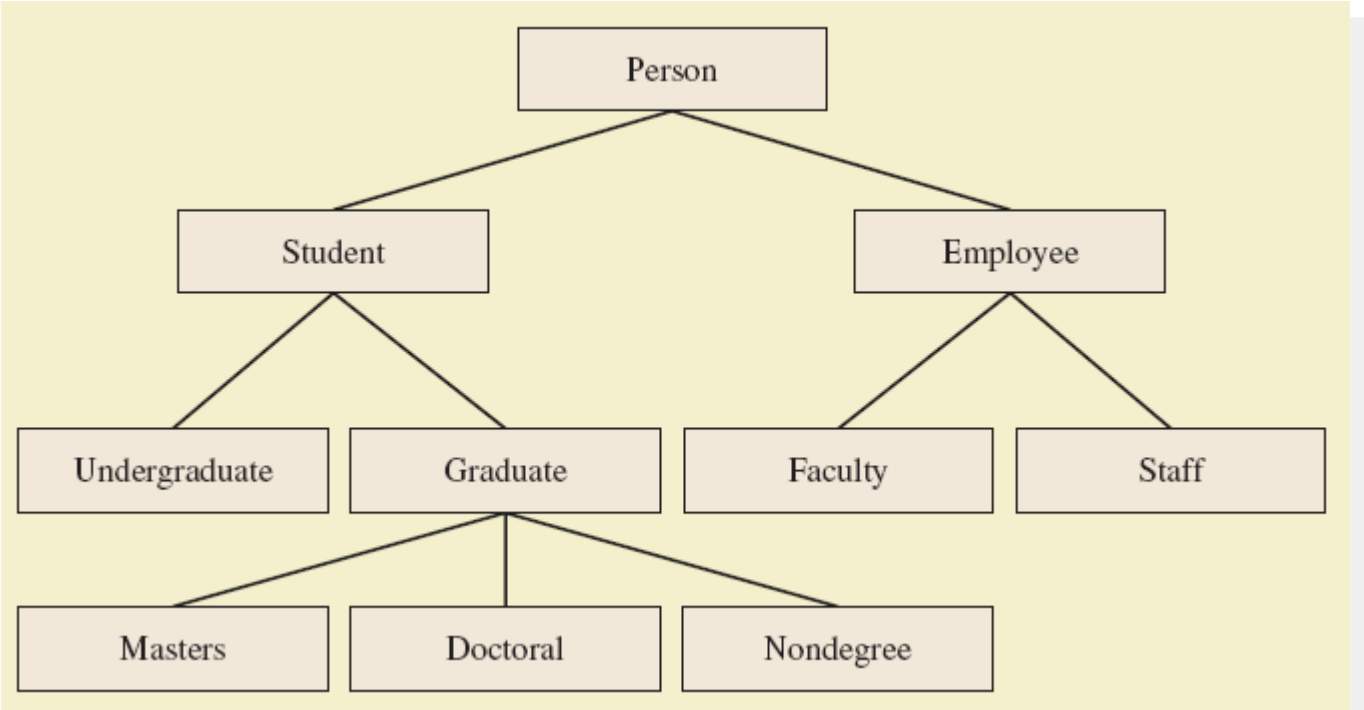
- Define the base class
- Define the subclass(es)
  - In the subclass, the class should be defined as:

    public <subclass name> extends <superclass name> {…}
  - The subclass constructor must call the superclass constructor in the first statement
    - If it does not and the superclass has a *default*, no argument constructor, the compiler will automatically insert it for you. (Error if default constructor not found)
    - Call superclass constructor with **super()**
    - Can call subclass constructor with **this()** if it eventually calls the superclass constructor
- Override any methods in the subclass that should override the superclass implementation
  - Can use super to call superclass implementation
  - Be careful about parameters – you may accidentally overload (same method name) when you mean to override (redefine existing method for subclass)

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Derived Class Example

- Figure 8.1 A class hierarchy (we do subtree of this)



ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Overriding Method Definitions

- Note method **`writeOutput`** in class **`Student`**
  - Class Person also has method with that name
- Method in subclass with same signature overrides method from base class
  - Overriding method is the one used for objects of the derived class
- Overriding method must return same type of value
- Do not confuse overriding with overloading
  - Overriding takes place in subclass – new method with same signature
  - Overloading -- New method in same class with different signature (possible one or more inherited)

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

The `final` Modifier

- Possible to specify that a method cannot be overridden in subclass
- Add modifier final to the heading

```
public final void specialMethod()
```

- Try setting Person's **writeOutput()** to **final**
- An entire class may be declared `final`
  - Thus cannot be used as a base class to derive any other class – e.g. java.lang.Math can not be subclassed/inherited from because declared final

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

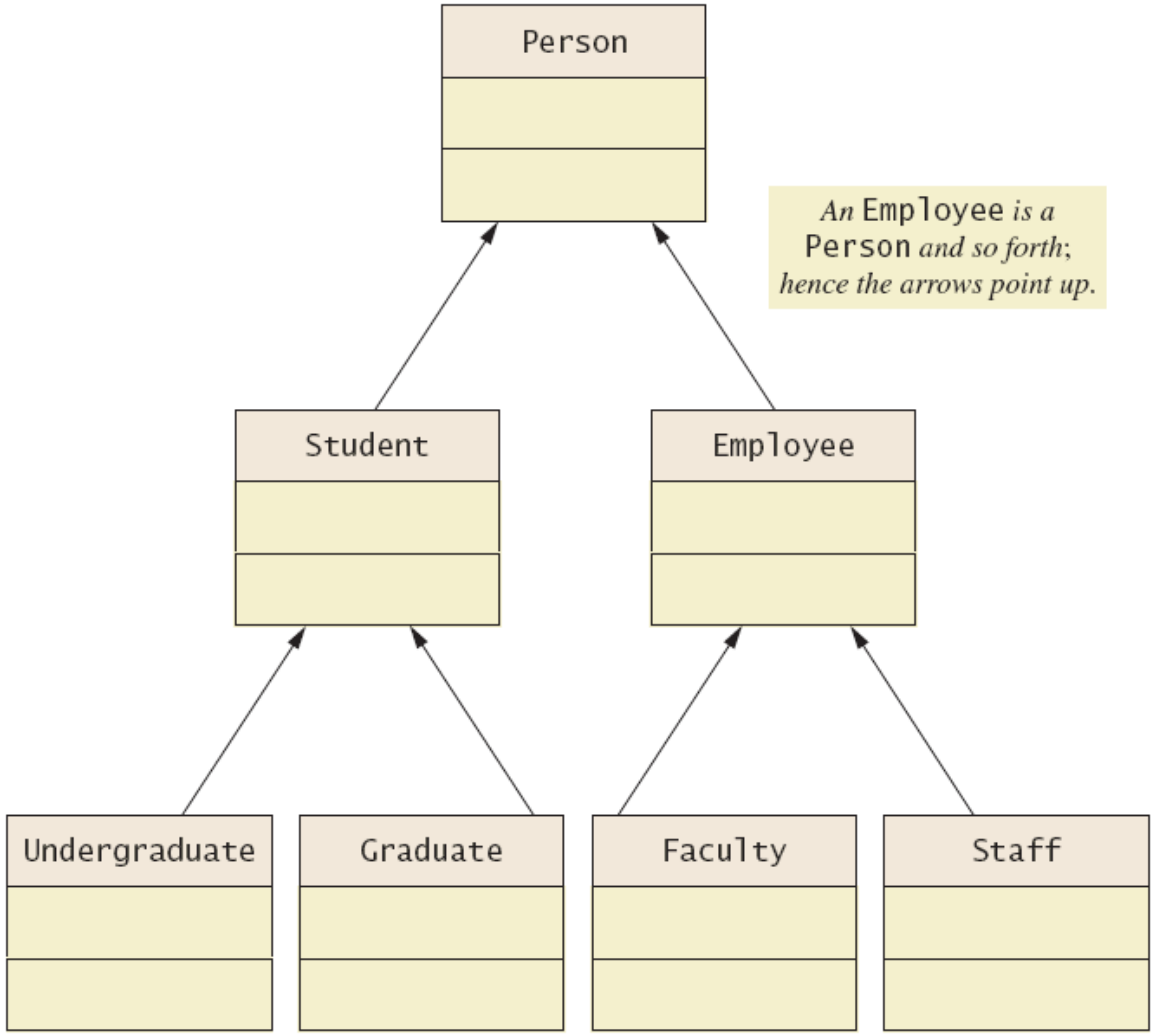
Private Instance Variables, Methods

- Consider private instance variable in a base class
  - It is not *accessible* in subclass
  - It can be manipulated only by public accessor, modifier methods
- Similarly, private methods in a superclass not *accessible* to subclass
- If you have some member that should not be public but should be accessible to subclasses, even ones outside the package, consider the access modifier **protected** (pg. 964, won't use for this course)

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

UML Inheritance Diagrams

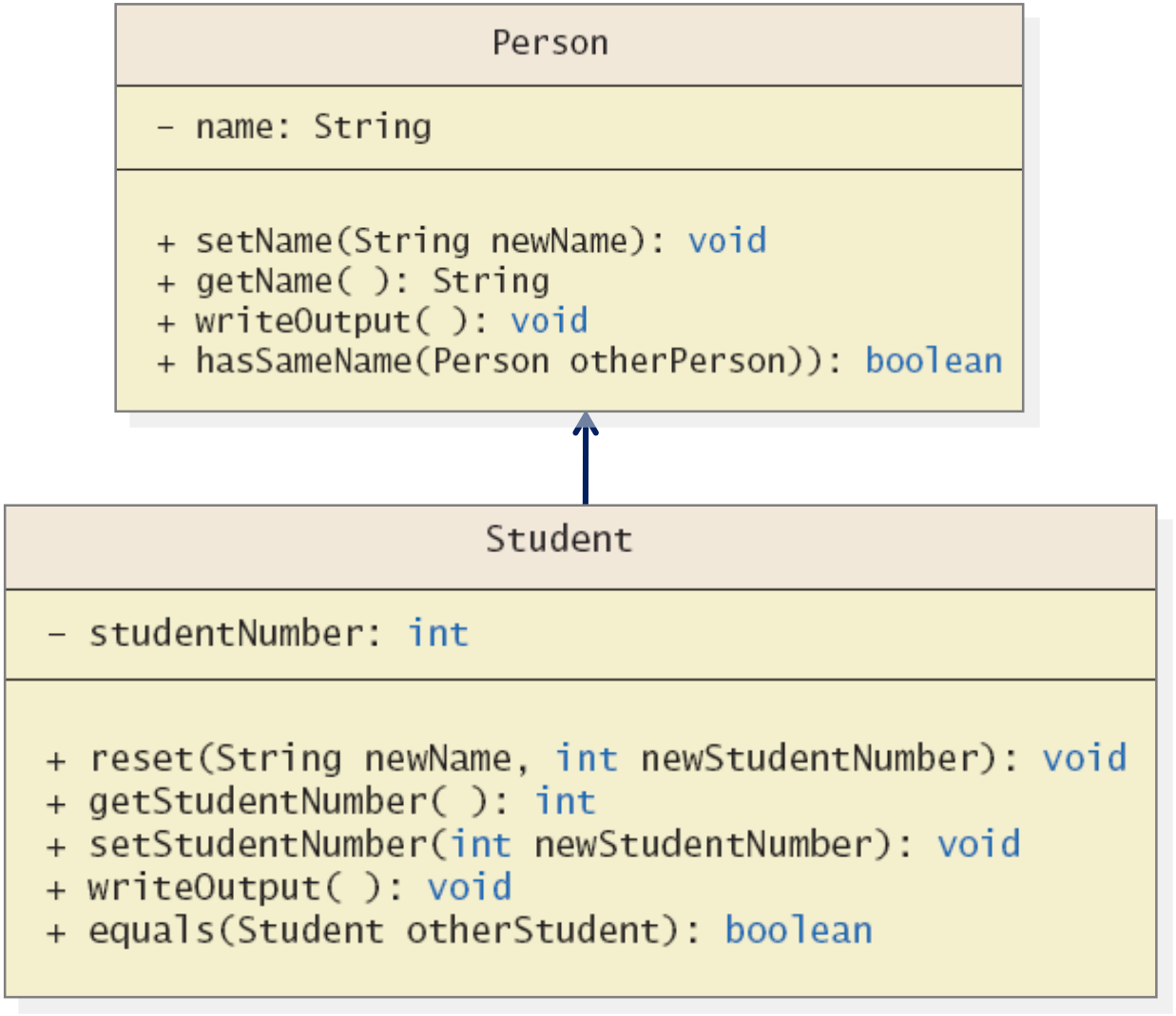
- Figure 8.2 A class hierarchy in UML notation



ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

UML Inheritance Diagrams

- Figure 8.3 Some details of UML class hierarchy from figure 8.2



ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Constructors in Derived Classes

- A derived class does not inherit constructors from base class
  - Constructor in a subclass must invoke constructor from base class
- Use the reserve word **`super`**

  - Must be first action in the constructor

```
public Student(String initialName, int initialStudentNumber)
{
    super(initialName);
    studentNumber = initialStudentNumber;
}
```


ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

The `this` Method – Again

- Also possible to use the **`this`** keyword
  - Use to call any constructor in the class

```
public Person()
{
    this("No name yet");
}
```

- When used in a constructor, this calls constructor in same class
  - Contrast use of **`super`** which invokes constructor of base class

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Calling an Overridden Method

- Reserved word **`super`** can also be used to call method in overridden method

```
public void writeOutput()
{
    super.writeOutput(); //Display the name
    System.out.println("Student Number: " + studentNumber);
}
```

- Calls method by same name in base class

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

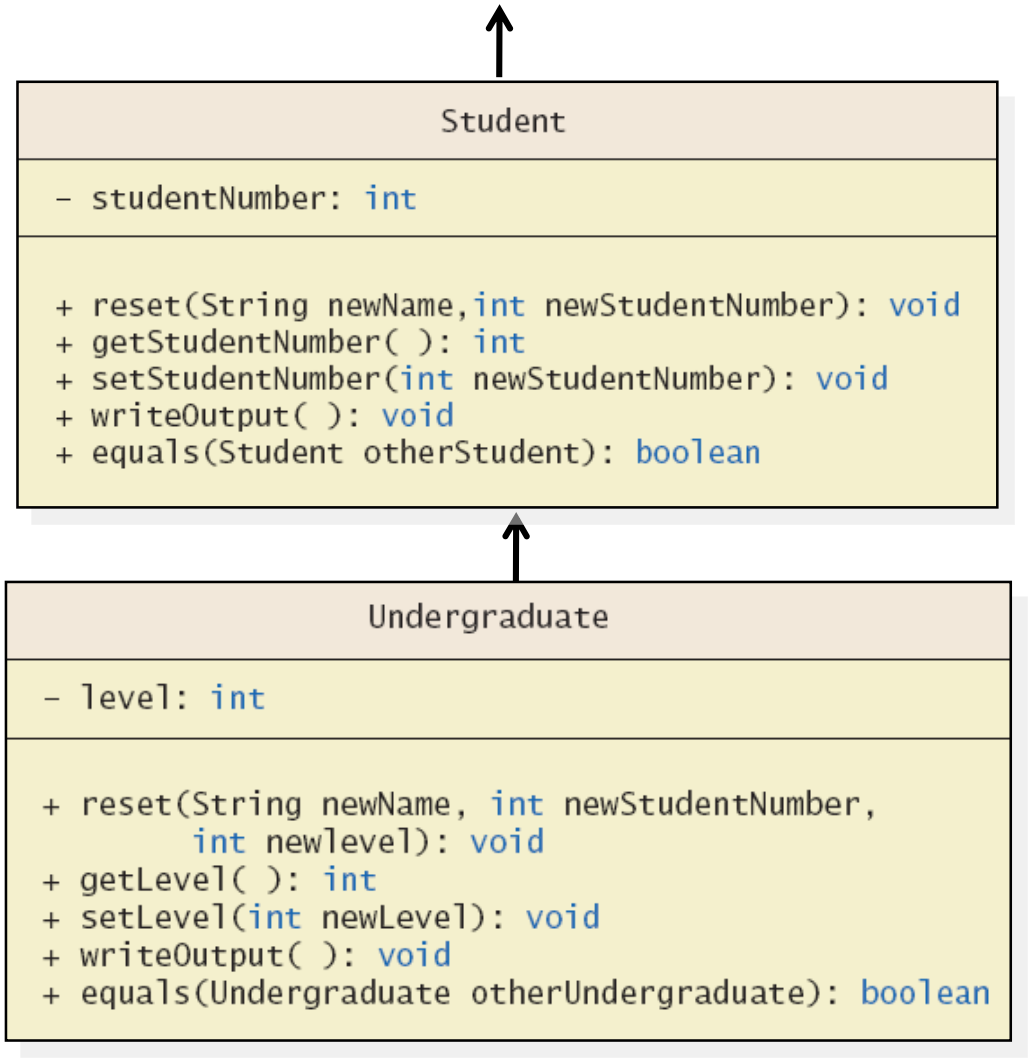
Programming Example

- A derived class of a derived class
- View sample class, listing 8.4
  `class Undergraduate`
- Has all public members of both
  - **`Person`**
  - **`Student`**
- This reuses the code in superclasses

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Programming Example

- Figure 8.4 More details of the UML class hierarchy



ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Inheritance is for **is-a** Relationships

- Be aware of the "is-a" relationship
  - A **`Student`** *is a* **`Person`**
- Another relationship is the "has-a"
  - A class can contain (as an instance variable) an object of another type
  - If we specify a date of birth variable for **`Person`** – it "has-a" **`Date`** object

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Type Compatibility

- In the class hierarchy
  - Each **`Undergraduate`** is also a **`Student`**
  - Each **`Student`** is also a **`Person`**
- An object of a derived class can serve as an object of the base class
  - Note this is not typecasting
- An object of a class can be referenced by a variable of an ancestor type (i.e. superclass variable can reference subtype instance, *but not the other way around*)

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

The Class `Object`

- Java has a class that is the ultimate ancestor of every class
  - The class `Object`
- Thus possible to write a method with parameter of type `Object`
  - Actual parameter in the call can be object of any type
- Example: method
  `println(Object theObject)`

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

The Class `Object`

- Class Object has some methods that every Java class inherits
- Examples (these are often *overridden*)
  - Method **`equals`** –this is same as **==** unless overridden
  - Method **`toString`** –default generally not very useful
- Method **`toString`** called when **`println(theObject)`** invoked
  - Best to define your own **`toString`** to handle this

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

A Better `equals` Method

- Programmer of a class should override method equals from `Object`
- Important detail we've been ignoring:
  - equals() takes an **Object** parameter
  - …not a member of the same class
  - If we specify same class as class declared – what is this called?
- View code of sample override, listing 8.8

```
public boolean equals
        (Object theObject)
```


ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Polymorphism

- Inheritance allows you to define a base class and derive classes from the base class
- Polymorphism allows you to make changes in the method definition for the derived classes and have those changes apply to methods written in the base class

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Polymorphism

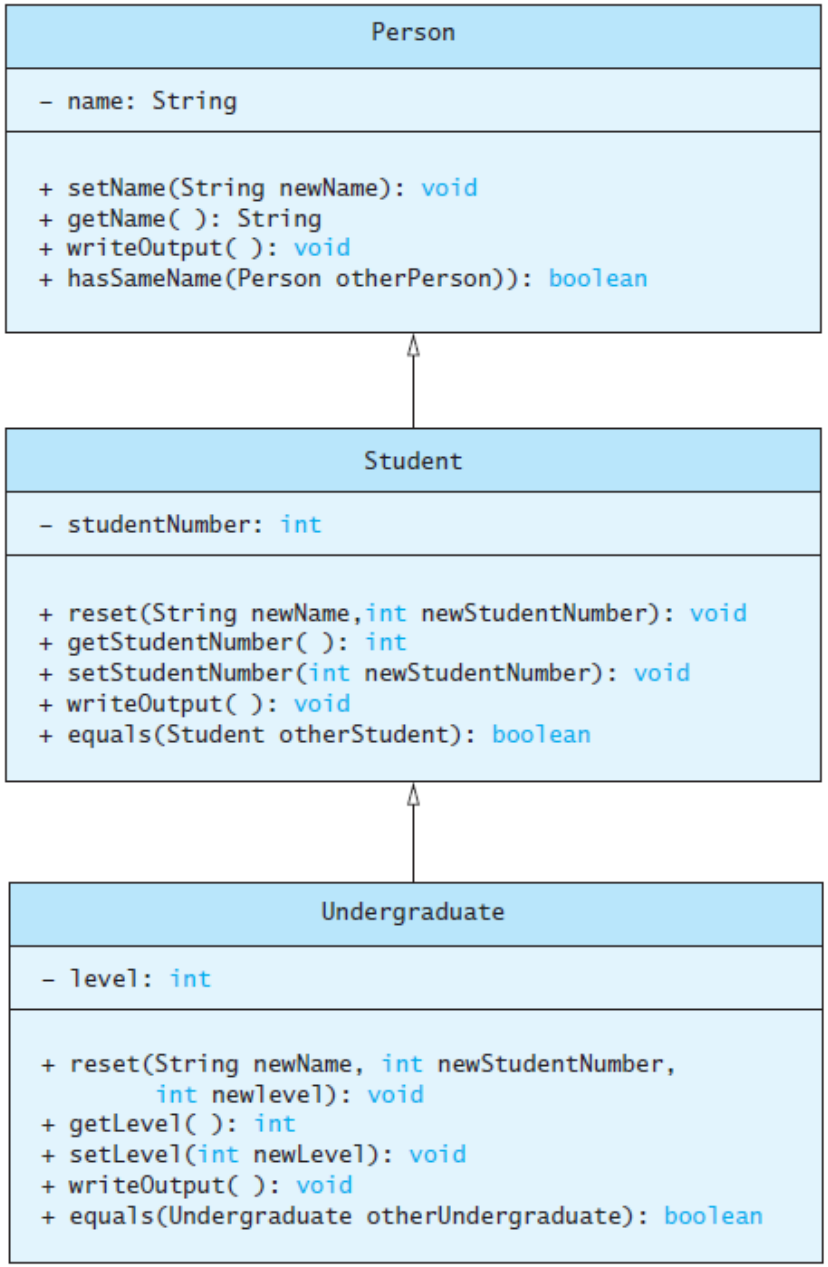
- Consider an array of **Person**

```
Person[] people = new Person[4];
```

- Since **Student** and **Undergraduate** are types of **Person**, we can assign them to **Person** variables

```
people[0] = new
Student("DeBanque, Robin",
8812);
```

```
people[1] = new
Undergraduate("Cotty, Manny",
8812, 1);
```



ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Polymorphism

- Given:

```
Person[] people = new Person[4];
people[0] = new Student("DeBanque, Robin",
8812);
```

- When invoking:

```
people[0].writeOutput();
```

- Which `writeOutput()` is invoked, the one defined for `Student` or the one defined for `Person`?
- Answer: The one defined for `Student`

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

An Inheritance as a Type

- The method can substitute one object for another
  - Called *polymorphism*
- This is made possible by mechanism
  - *Dynamic binding*
  - Also known as *late binding*

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Dynamic Binding and Inheritance

- When an overridden method invoked
  - Action matches method defined in class used to create object using **`new`**, i.e. the actual constructor called.
  - *Not* determined by type of *variable* referencing the object
- Variable of any ancestor class can reference object of descendant class
  - Object always "remembers" which method actions to use for each method name

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Polymorphism Example

- View sample class, listing 8.6
  `class PolymorphismDemo`
- Output

```
Name: Cotty, Manny
Student Number: 4910
Student Level: 1

Name: Kick, Anita
Student Number: 9931
Student Level: 2
```

```
Name: DeBanque, Robin
Student Number: 8812

Name: Bugg, June
Student Number: 9901
Student Level: 4
```


ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Class Interfaces

- Consider a set of behaviors for pets
  - Be named
  - Eat
  - Respond to a command
- We could specify method headings for these behaviors
- These method headings can form a class interface

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Class Interfaces

- Now consider different classes that implement this interface
  - They will each have the same behaviors
  - Nature of the behaviors will be different
- Each of the classes implements the behaviors/methods differently
- An interface is a different type of **is-a** (or is) relationship: instead of inheriting features, it is contract-based -- a class that implements an interface provides set of methods, generally, that satisfy the requirements to be treated as a one of the "interface" type
- Side Note: Java does not allow multiple inheritance but a class can implement multiple intefaces (can do both at the same time too)

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Java Interfaces

- Interface name begins with uppercase letter
- Stored in a file with suffix **`.java`**
- Interface does not include
  - Declarations of constructors
  - Instance variables
  - Method bodies

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Implementing an Interface

- To implement a method, a class must
  - Include the phrase

    `implements Interface_name`
  - Define each specified method
- View sample class, listing 8.8

  `class Rectangle implements Measurable`
- View another class, listing 8.9 which also implements Measurable

  class Circle

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

An Interface as a Type

- Possible to write a method that has a parameter as an interface type
  - An interface is a reference type
- Program invokes the method passing it an object of any class which implements that interface
- Possible to define a new interface which builds on an existing interface
  - It is said to extend the existing interface
- A class that implements the new interface must implement all the methods of both interfaces

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Case Study

- Character “Graphics”
- View interface for simple shapes,
  listing 8.10 **`interface ShapeInterface`**
- If we wish to create classes that draw rectangles and triangles
  - We could create interfaces that extend **`ShapeInterface`**
  - View interfaces, listing 8.11

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

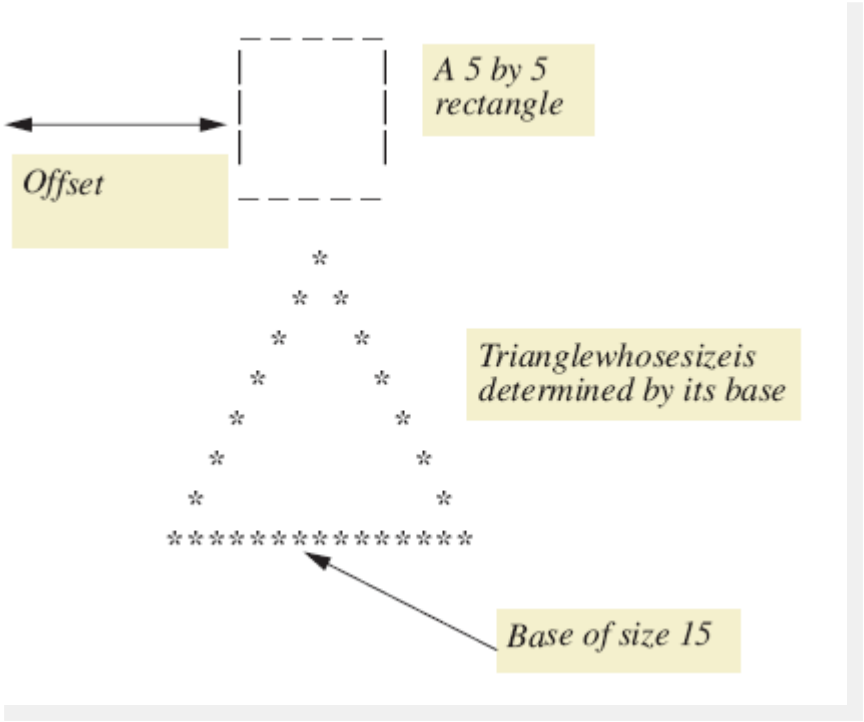
Case Study

- Now view base class, listing 8.12 which uses (implements) previous interfaces
  `class ShapeBasics`
- Note
  - Method **`drawAt`** calls **`drawHere`**
  - Derived classes must override **`drawHere`**
  - Modifier **`extends`** comes before **`implements`**

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Case Study

- Figure 8.5 A sample rectangle and triangle



ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Case Study

- Note algorithm used by method **`drawHere`** to draw a rectangle
  1. Draw the top line
  2. Draw the side lines
  3. Draw the bottom lines
- Subtasks of **`drawHere`** are realized as private methods
- View class definition, listing 8.13
  `class Rectangle`

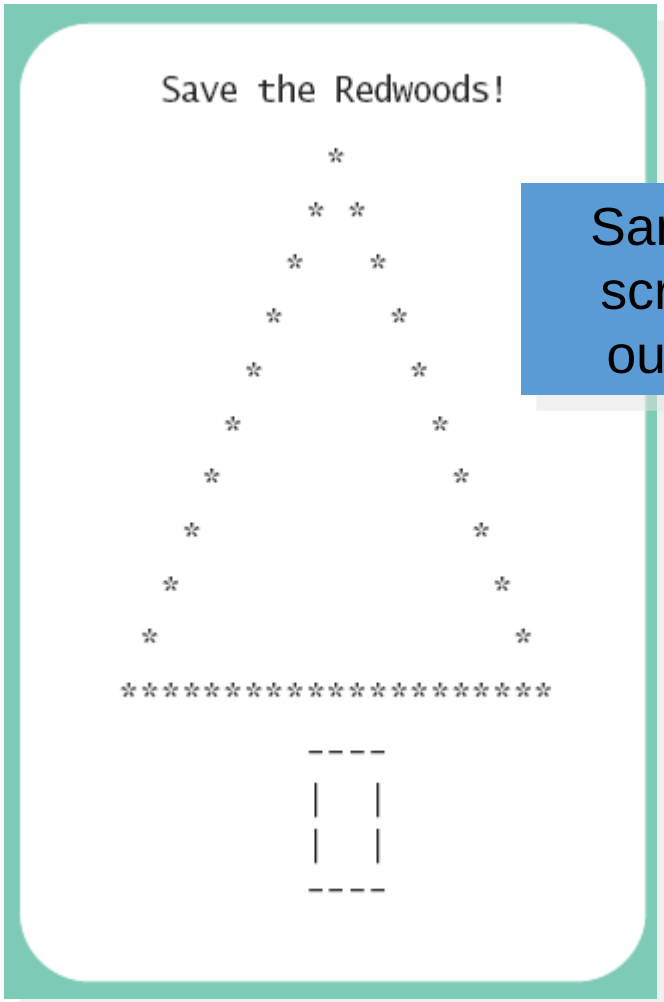
ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Case Study

- View next class to be defined (and tested),
  listing 8.14 class Triangle
- It is a good practice to test the classes as we go
- View demo program, listing 8.15
  `class TreeDemo`

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Case Study



Sample screen output

*JAVA: An Introduction to Problem Solving & Programming, 7*th Ed. By Walter Savitch
ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Case Study
The `Comparable` Interface

- Java has many predefined interfaces
- One of them, the `Comparable` interface, is used to impose an ordering upon the objects that implement it
- Requires that the method `compareTo` be written

```
public int compareTo(Object other);
```


ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Sorting an Array of Fruit Objects

- Initial (non-working) attempt to sort an array of **`Fruit`** objects
- View class definition, listing 8.16
  `class Fruit`
- View test class, listing 8.17
  `class FruitDemo`
- Result: Exception in thread "main"
  - Sort tries to invoke **`compareTo`** method but it doesn't exist

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Sorting an Array of Fruit Objects

- Working attempt to sort an array of **`Fruit`** objects – implement **`Comparable`**, write **`compareTo`** method
- View class definition, listing 8.18
  `class Fruit`
- Result: Exception in thread "main"
  - Sort tries to invoke method but it doesn't exist

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

`compareTo` Method

- An alternate definition that will sort by length of the fruit name

```
public int compareTo(Object o)
{
    if ((o != null) &&
        (o instanceof Fruit))
    {
         Fruit otherFruit = (Fruit) o;
         if (fruitName.length() >
             otherFruit.fruitName.length())
             return 1;
         else if (fruitName.length() <
                  otherFruit.fruitName.length())
               return -1;
         else
               return 0;
    }
         return -1;  // Default if other object is not a Fruit
}
```


ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Abstract Classes

- Class **`ShapeBasics`** is designed to be a base class for other classes
  - Method **`drawHere`** will be redefined for each subclass
  - It could be declared *abstract* – a method that has no body
- This makes the class abstract
- Some things too nebulous, like "shape", to create an actual instance of
- You cannot create an object of an abstract class – thus its role as base class
- Not all methods of an abstract class are necessarily abstract methods
- Abstract class makes it easier to define a base class
  - Specifies the obligation of subclass writer to override the abstract methods for each subclass

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Abstract Classes (basic recipe)

- Cannot have an instance of an abstract class
  - But OK to have a parameter variable of that type that will reference a subtype (that is *not* abstract) of the abstract class
- View CaseStudyAbstract
- Declare class abstract

```
public abstract class ShapeBase implements ShapeInterface {
```

- Add abstract to any methods that are abstract. Make "empty" (no body, i.e. **{ }** – just semicolon)

```
public abstract void drawHere();
```

- Override the abstract method in the subclasses to get class that can be instantiated – inheritance works as normal (traits inherited from abstract to normal)
- If you don't override all abstract methods, the class must be declared abstract: see **ShapeBaseSubclassStillAbstract**

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Dynamic Binding and Inheritance

- Note how **`drawAt`** (in **`ShapeBasics`**) makes a call to **`drawHere`**
- Class **`Rectangle`** overrides method **`drawHere`**
  - How does **`drawAt`** know where to find the correct **`drawHere`**?
- Happens with dynamic or late binding
  - Address of correct code to be executed determined at run time based on actual type of object, not the variable referencing it (methods have addresses too)

ISBN 0133862119 © 2015 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved