Object Oriented Programming

Introduction to Java

Ch. 8. Inheritance, Polymorphism and Interfaces

Dept. of Software, Gachon University
Ahyoung Choi, Spring

8.1 Inheritance Basics

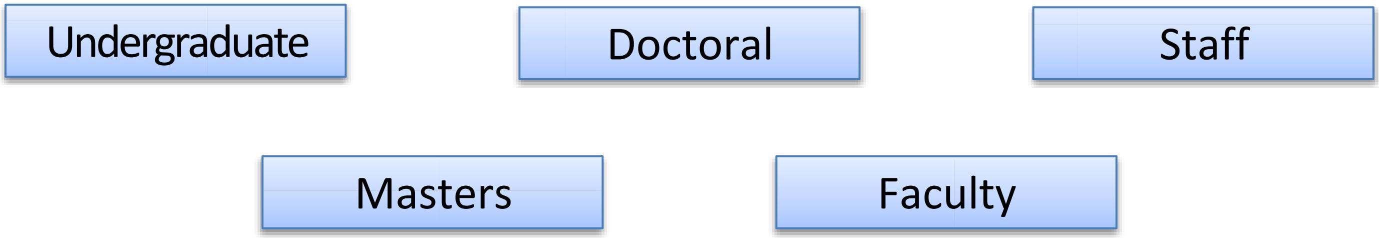
Inheritance

- Important questions:
  - What is inheritance?
  - How to use inheritance?

- The biggest difficulty:
  - Inheritance is specifically used for “better design”
  - **Design** is harder than implementation, so you haven’t done much design

Motivation

- Suppose designing a college record-keeping program
  - What types of data are required?
- Many classes share the same fields and methods
  - E.g., name, phone number, …
  - You should not copy & paste the duplicate variables & methods
- It is good to reuse codes



Undergraduate

Doctoral

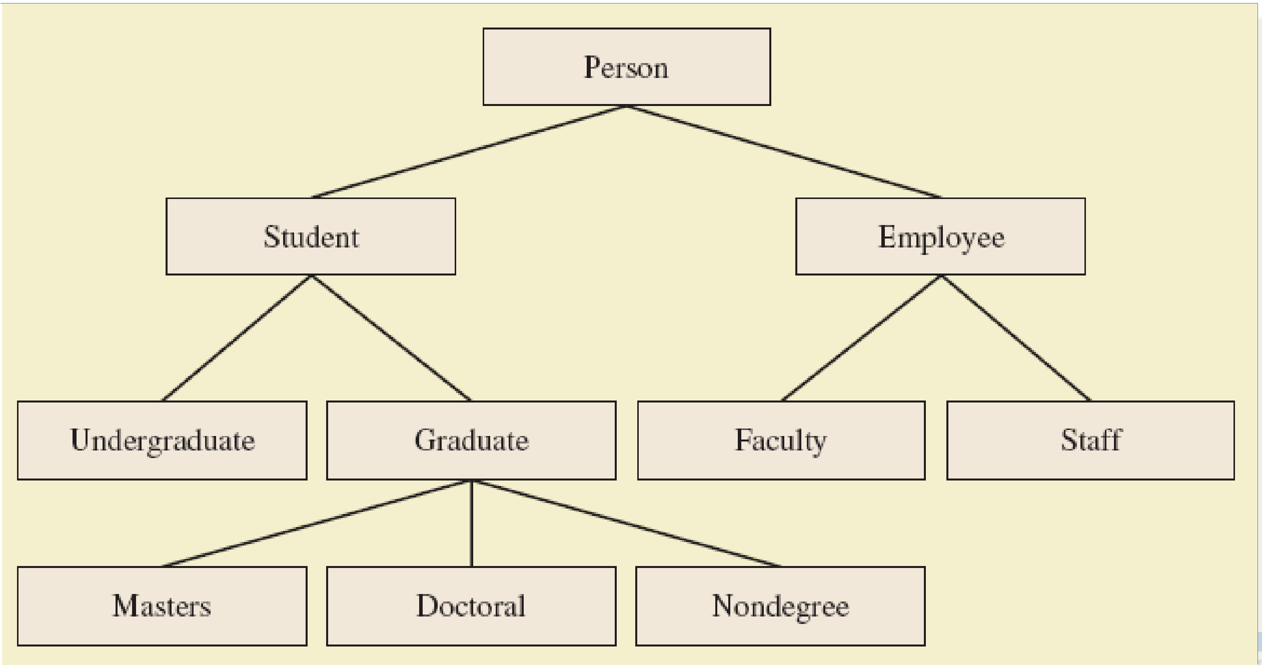
Staff

Masters

Faculty

Class hierarchy

- A way to organize classes
- *Derived classes* share the characteristics of *base class*
  - Inherit variables and methods from base class
  - Also referred to as *subclass* and *superclass*



Syntax Rules

public class **Derived_Class_Name** **extends** **Base_Class_Name**

- `public class MountainBike extends Bicycle`

- After the inheritance, the subclass inherits all the **public** variables and methods of the superclass
- Also, the subclass can add new variables and methods
  - Bicycle class has *cadence, gear, speed*, constructor and four setters
  - MountainBike class has ***cadence, gear, speed***, seatHeight, constructor, **four setters** and a new setter setHeight()

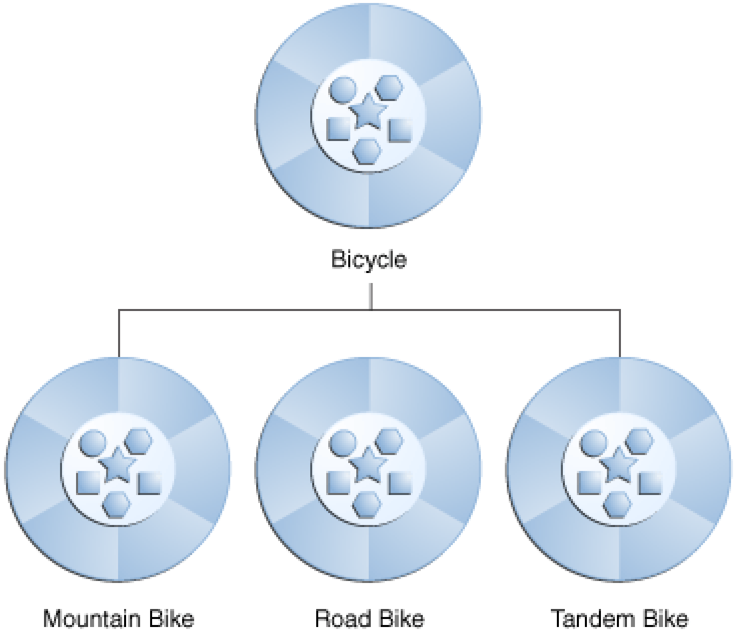
Syntax Rules

- Private members
  - Private instance variables in a superclass are not inherited in subclass
    - It can be accessed/modified only with public accessor/mutator methods in the superclass
  - Similarly, private methods in a superclass are not inherited in subclass

Access modifier	Same class	Same package	Subclass	Other
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
private	✓	✗	✗	✗

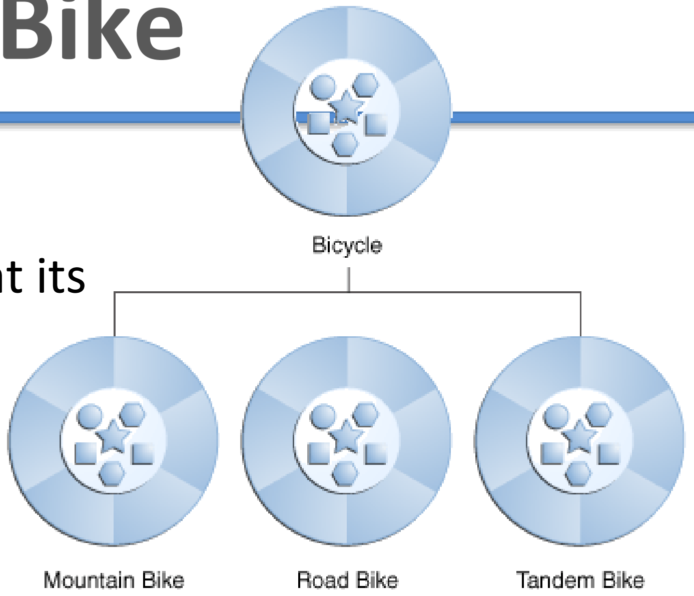
Example: class Bike

```java
public class Bicycle {
    // the Bicycle class has three fields
    public int cadence, gear, speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear; cadence = startCadence; speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }
    public void speedUp(int increment) {
        speed += increment;
    }
}
```



Example: class MountainBike

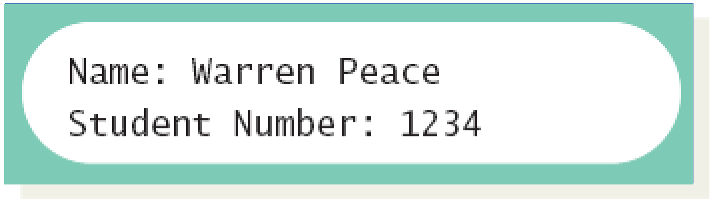
- **MountainBike**
  - All properties are same with Bike except that its seatHeight is adjustable



```java
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed,
        int startGear) {
        super(startCadence, startSpeed, startGear); // introduce later
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

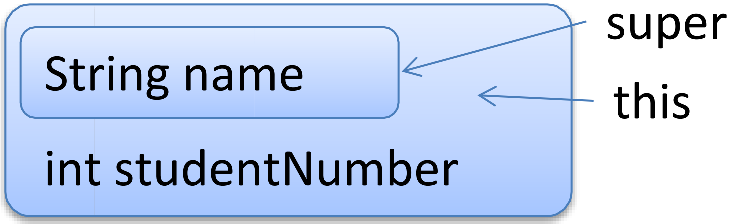
Lab: Derived Classes

- View derived class, listing 8.2
  class Student extends Person
- View demo program, listing 8.3
  class InheritanceDemo



```
Name: Warren Peace
Student Number: 1234
```

Lab



Superclass

```
public class Person {
    private String name;
    public Person() {
        name = "No name yet";
    }
    public Person(String initialName) {
        name = initialName;
    }
    public void setName(String newName) {
        name = newName;
    }
    public String getName() {
        return name;
    }
    public void writeOutput() {
        System.out.println("Name:  " + name);
    }
    public boolean hasSameName(Person otherPerson) {
        return 
this.name.equalsIgnoreCase(otherPerson.name);
    }
}
```

Subclass

```
public class Student extends Person {
    private int studentNumber;
    public Student() {
        super();
        studentNumber = 0;
    }
    public Student(String initialName, int 
initialStudentNumber) {
        super(initialName);
        studentNumber = initialStudentNumber;
    }
    public void reset(String newName, int 
newStudentNumber) {
        setName(newName);
        studentNumber = newStudentNumber;
    }
    public int getStudentNumber() {
        return studentNumber;
    }
    public void setStudentNumber(int 
newStudentNumber) {
        studentNumber = newStudentNumber;
    }
```

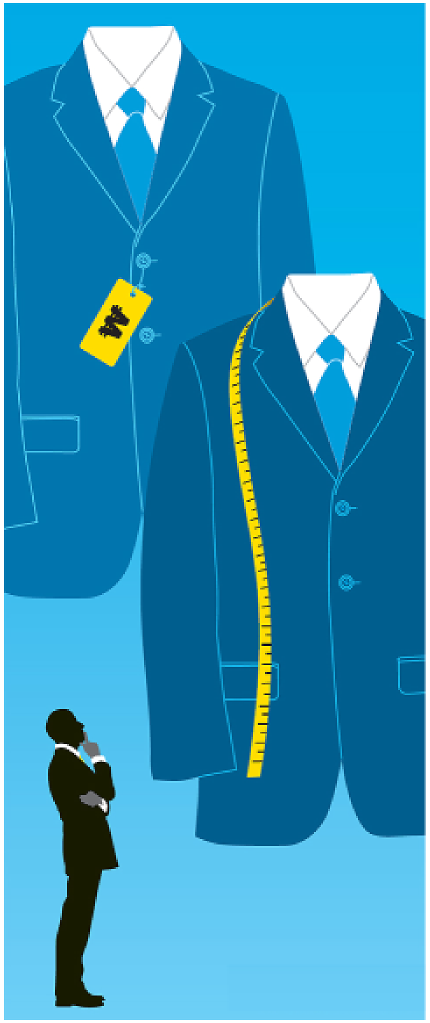
Lab: test class

```
public class InheritanceDemo
{
   public static void main (String [] args)
   {
      Student s = new Student ();
      s.setName ("Warren Peace");
      s.setStudentNumber (1234);
      s.writeOutput ();
   }
}
```

// we did not define setName() in the class Student

Question

- What if you want to **customize** some methods of the superclass ?



If want to tune openTrunk() method?

Superclass
(base class)

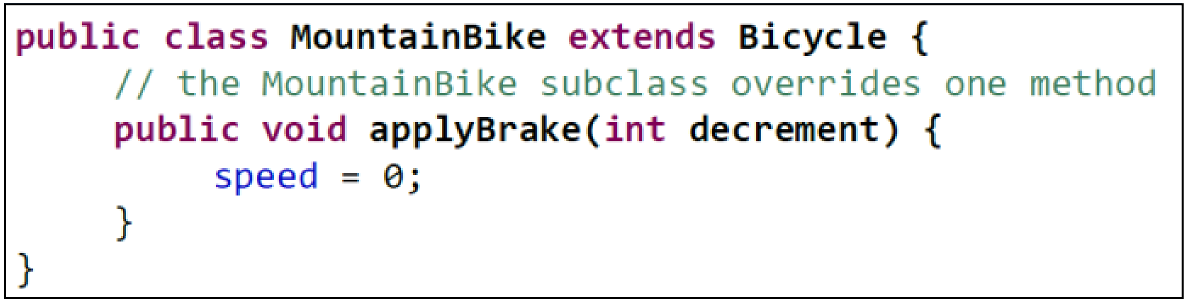
openTrunk()

Inherit

openTrunk()

More Inheritance: Override

- You can write a method (and variables) in the subclass to **rewrite/replace** the method **with the same name** in the superclass
  - Note method `writeOutput` in class `Student` **(listing 8.2)**
    - Class Person also has method with that name
  - For example, the MountainBike has a powerful break so it immediately reduce the speed to 0



```
public class MountainBike extends Bicycle {
     // the MountainBike subclass overrides one method
     public void applyBrake(int decrement) {
          speed = 0;
     }
}
```

- Now if we call mb.applyBrake(3), the speed will be 0

Overriding Method Definitions

- 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

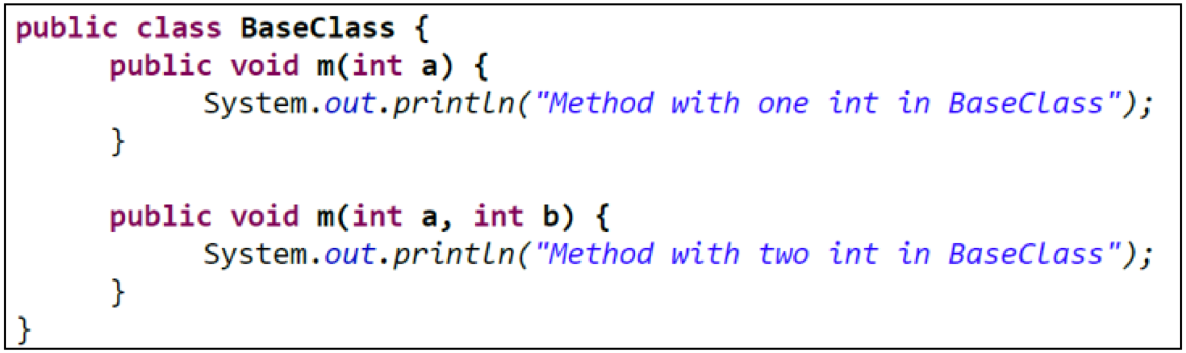
Overriding vs Overloading

- Overriding vs. overloading
  - Overriding: a method in subclass with the same signature
  - Overloading: methods with the same name and different parameters in the same class

```
public class BaseClass {
    public void m(int a) {
        System.out.println("B1");
    }
    public void m(int a, int b) {
        System.out.println("B2");
    }
}
```

```
public class DerivedClass extends BaseClass {
    @Override
    public void m(int a) {
        System.out.println("D1");
    }
    public static void main(String[] args) {
        DerivedClass b = new DerivedClass();
        b.m(0);          // D1
        b.m(0, 0);       // B2
    }
}
```

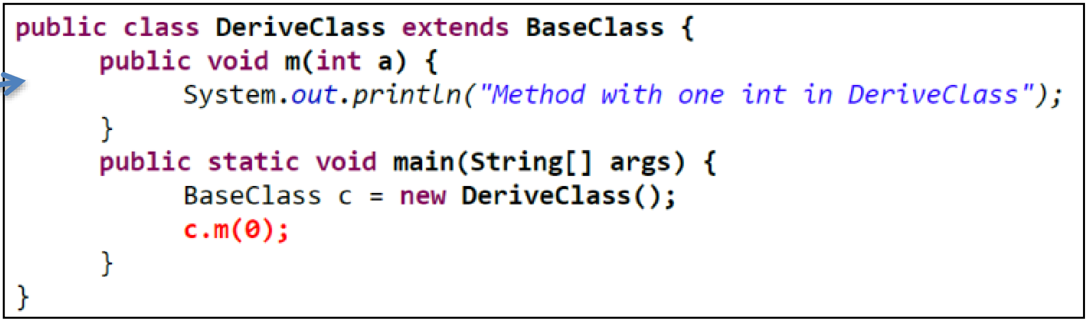
Overriding vs Overloading



```java
public class BaseClass {
	public void m(int a) {
		System.out.println("Method with one int in BaseClass");
	}

	public void m(int a, int b) {
		System.out.println("Method with two int in BaseClass");
	}
}
```

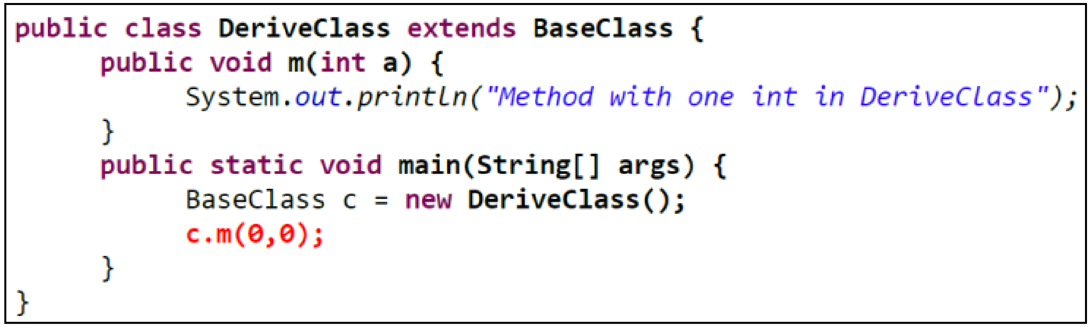
Overloading!

Overriding!



```java
public class DeriveClass extends BaseClass {
	public void m(int a) {
		System.out.println("Method with one int in DeriveClass");
	}
	public static void main(String[] args) {
		BaseClass c = new DeriveClass();
		c.m(0);
	}
}
```

Output ?



```java
public class DeriveClass extends BaseClass {
	public void m(int a) {
		System.out.println("Method with one int in DeriveClass");
	}
	public static void main(String[] args) {
		BaseClass c = new DeriveClass();
		c.m(0,0);
	}
}
```

Output ?

`final` Modifier

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

  `public final void specialMethod()`

- An entire class may be declared **`final`**
  - cannot be used as a base class to derive any other class

Private Instance Variables, Methods

- Consider private instance variable in a base class
  - It is not inherited in subclass (but! accessible)
  - It can be manipulated only by public accessor, modifier methods
- Similarly, private methods in a superclass not inherited by subclass

Problem ?

```
public class Person
{
    private String name;
 …
}

 public class Student extends Person
 {
   …
   public void reset(String newName, int newStudentNumber)
   {
         name = newName;
         studentNumber = newStudentNunmber;
   }
```

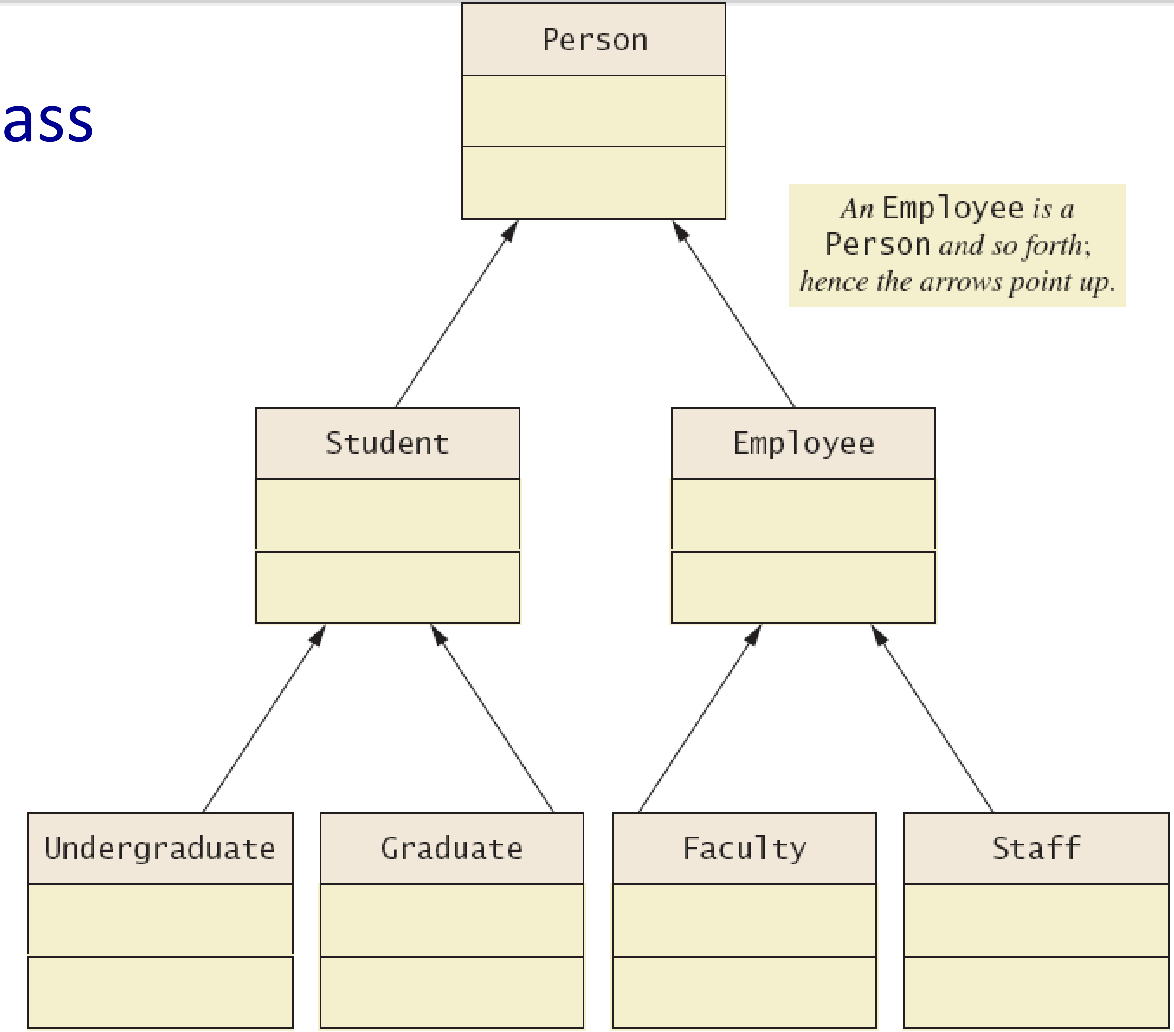
Example

```
public class Person
{
    private String name;
 …
}

 public class Student extends Person
 {
   …
   public void reset(String newName, int newStudentNumber)
   {
         // name = newName;  // ILLEGAL !
         setName(newName);    // valid !!
         studentNumber = newStudentNunmber;
   }
```

- The derived class does not inherit private variables. Thus, you should use mutator methods to set the value

UML Inheritance Diagrams

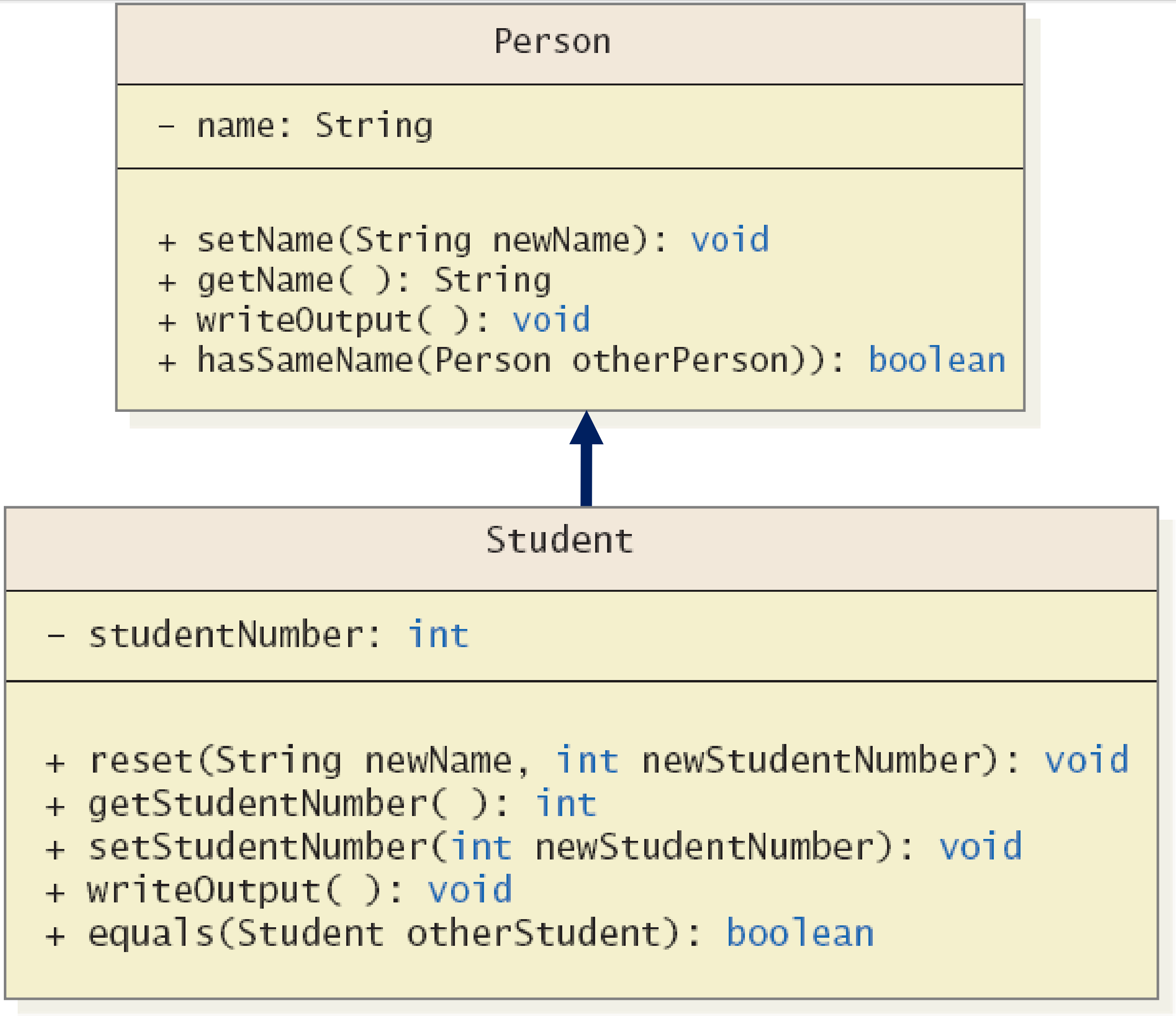
- Figure 8.2 A class hierarchy in UML notation



An Employee *is a* Person *and so forth; hence the arrows point up.*

UML Inheritance Diagrams

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



Person

- name: String

+ setName(String newName): void
+ getName( ): String
+ writeOutput( ): void
+ hasSameName(Person otherPerson)): boolean

Student

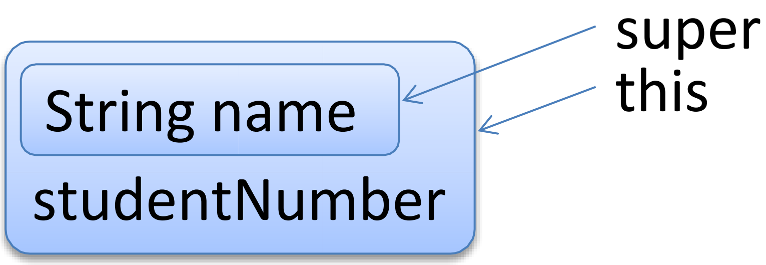
- studentNumber: int

+ reset(String newName, int newStudentNumber): void
+ getStudentNumber( ): int
+ setStudentNumber(int newStudentNumber): void
+ writeOutput( ): void
+ equals(Student otherStudent): boolean

Lab: Class Person and Student

- Override

```
    @Override
    public void writeOutput() {
        super.writeOutput();
        System.out.println("Student Number: " + studentNumber);
    }
    public boolean equals(Student otherStudent) {
        return this.hasSameName(otherStudent) &&
            (this.studentNumber == otherStudent.studentNumber);
    }
}
```



String name

studentNumber

super

this

8.2 Programming with Inheritance

Fields/Methods in Extended Classes

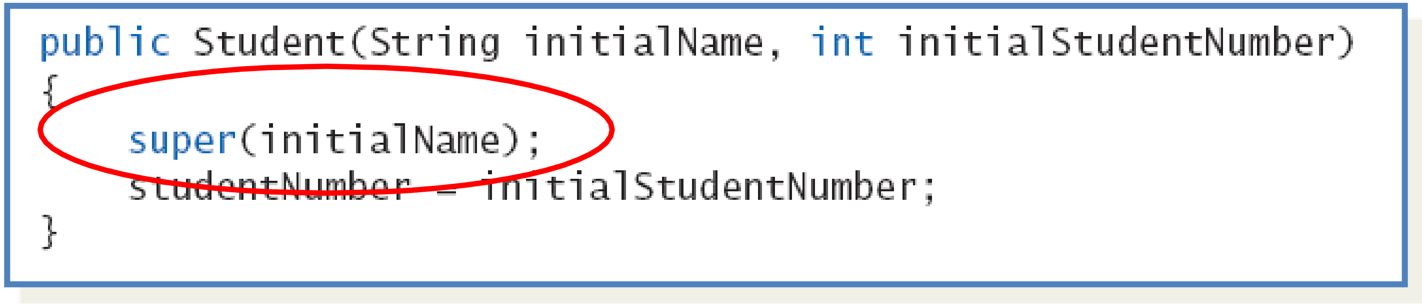
- An object of an extended class contains two sets of variables and methods
  - fields/methods which are defined locally in the extended class
  - fields/methods which are inherited from the superclass

- What are the fields for a Student object in the previous example

☞ How to initialize each set of the fields?

Constructors in Derived Classes

- A derived class does not inherit constructors from base class
  - Usually, the initialization of the base class is required
  - Constructor in a subclass must invoke constructor from base class
- Use the reserve word **super**

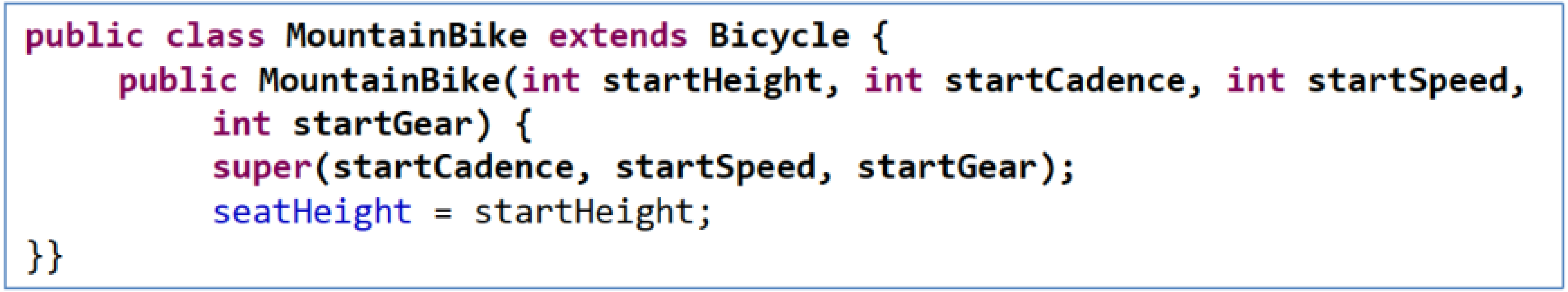


```
public Student(String initialName, int initialStudentNumber)
{
    super(initialName);
    studentNumber = initialStudentNumber;
}
```

  - Must be first action in the constructor

Using the Keyword *super*

- *super* can be used to invoke superclass's constructor.



```
public class MountainBike extends Bicycle {
    public MountainBike(int startHeight, int startCadence, int startSpeed,
        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
}}
```

- **Detail:**
  - **It must be the first line in the subclass constructor**
  - The default constructor super() will be automatically called if you do not include an explicit call to the base-class constructor
  - If the super class does not have a no-argument constructor, you **must** invoke the super class constructor with a matching parameter list

To Illustrate the Construction Order. . .

```
class X {
     protected int xOri = 1;
     protected int whichOri;
     public X() {
          whichOri = xOri;
     }
}
```

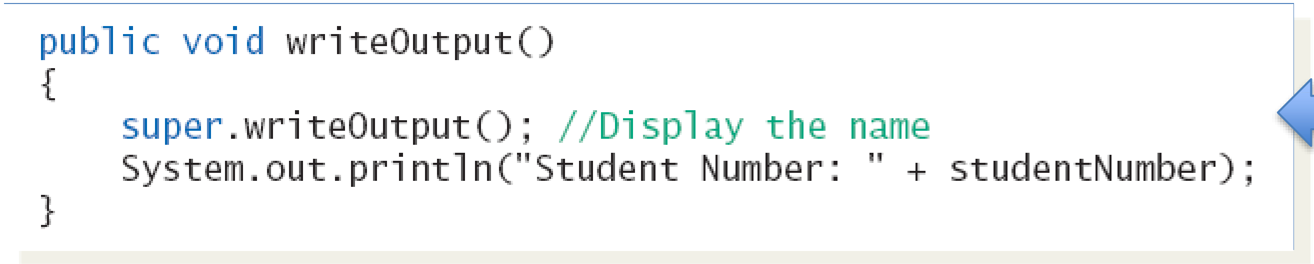
```
class Y extends X {
     protected int yOri = 2;
     public Y() {
          whichOri = yOri;
     }
}
```

```
Y objectY = new Y();
```

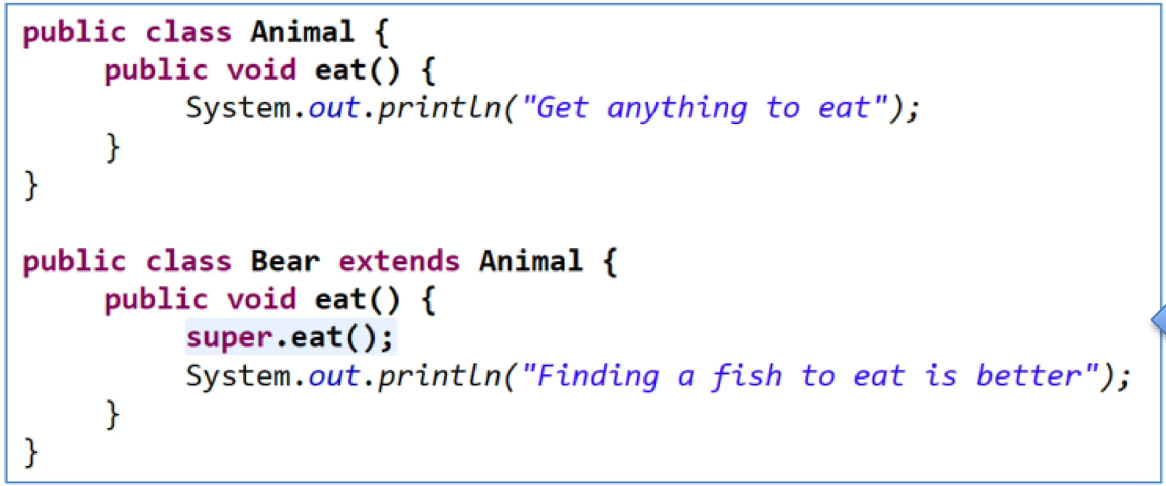
Step	what happens	xOri	yOri	whichOri
0	memory alloc & fields set to default values	0	0	0
1	Y constructor invoked	0	0	0
2	X constructor invoked	0	0	0
3	X field initialization	1	0	0
4	X constructor executed	1	0	1
5	Y field initialization	1	2	1
6	Y constructor executed	1	2	2

Calling an Overridden Method

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



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



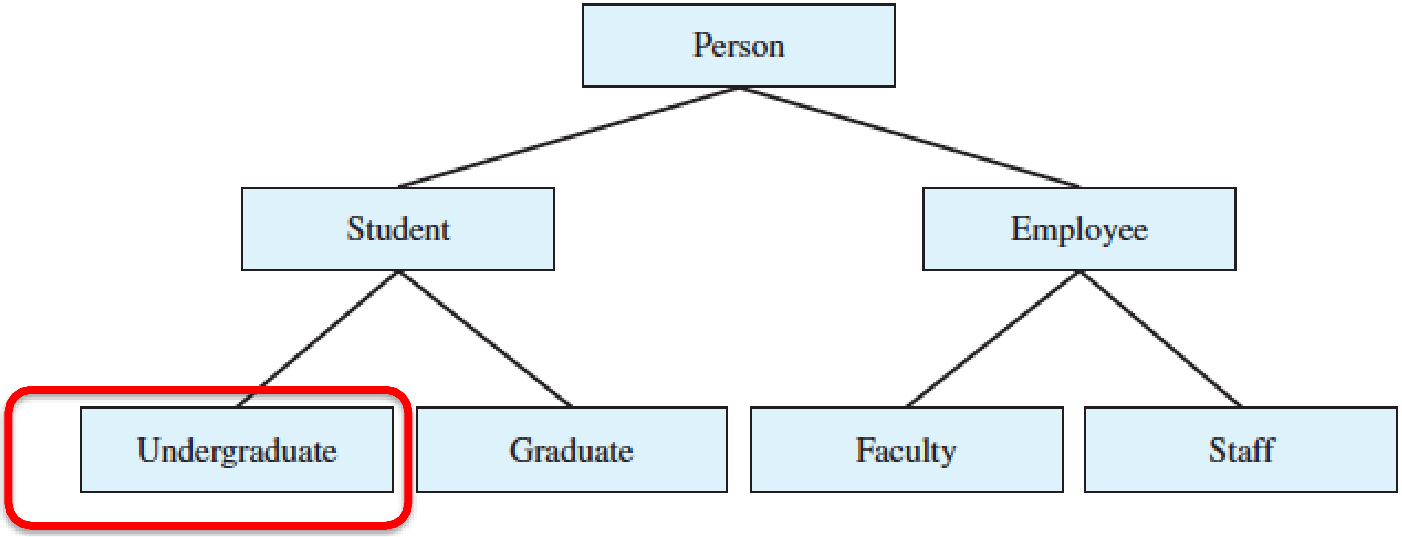
```java
public class Animal {
    public void eat() {
        System.out.println("Get anything to eat");
    }
}

public class Bear extends Animal {
    public void eat() {
        super.eat();
        System.out.println("Finding a fish to eat is better");
    }
}
```

- Calls method by same name in base class

A derived class of a derived class

(Listing 8.4) `class Undergraduate`

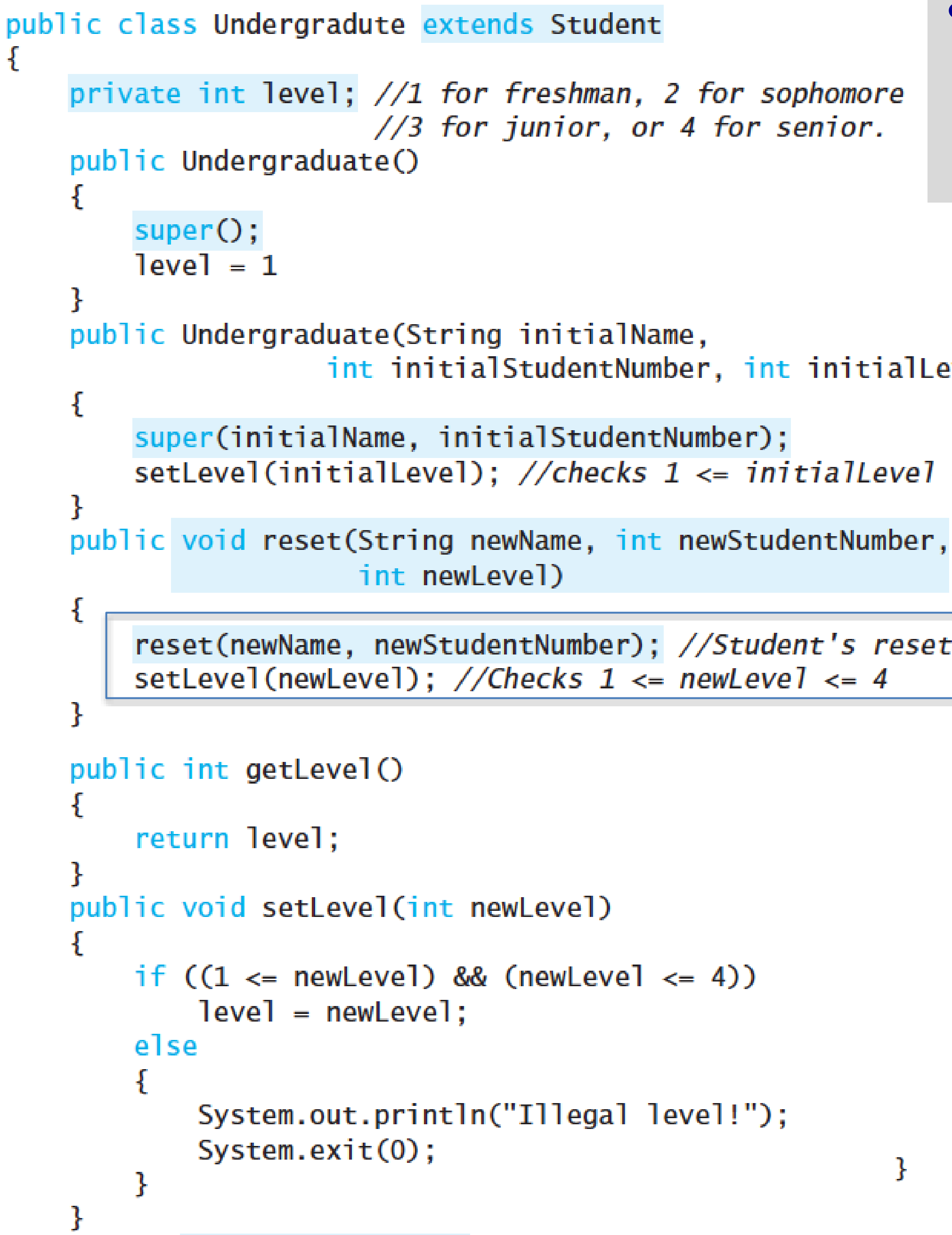
```
class Undergraduate extends Student
```



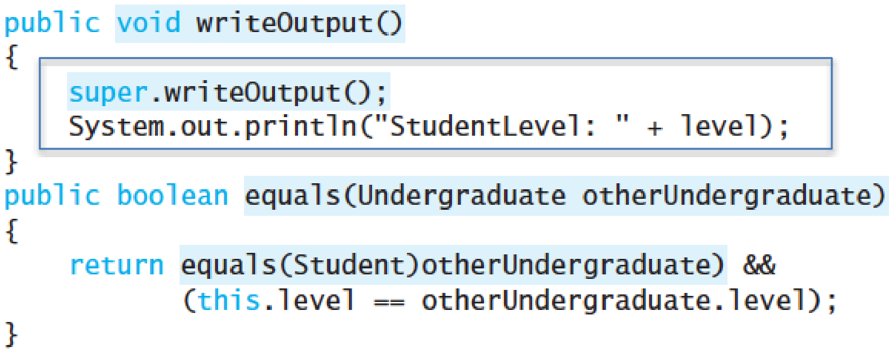
Undergraduate has

All the public members of the class **Student** and **Person**

- **Undergraduate** has
  - All the public members of the class **Student** and **Person**



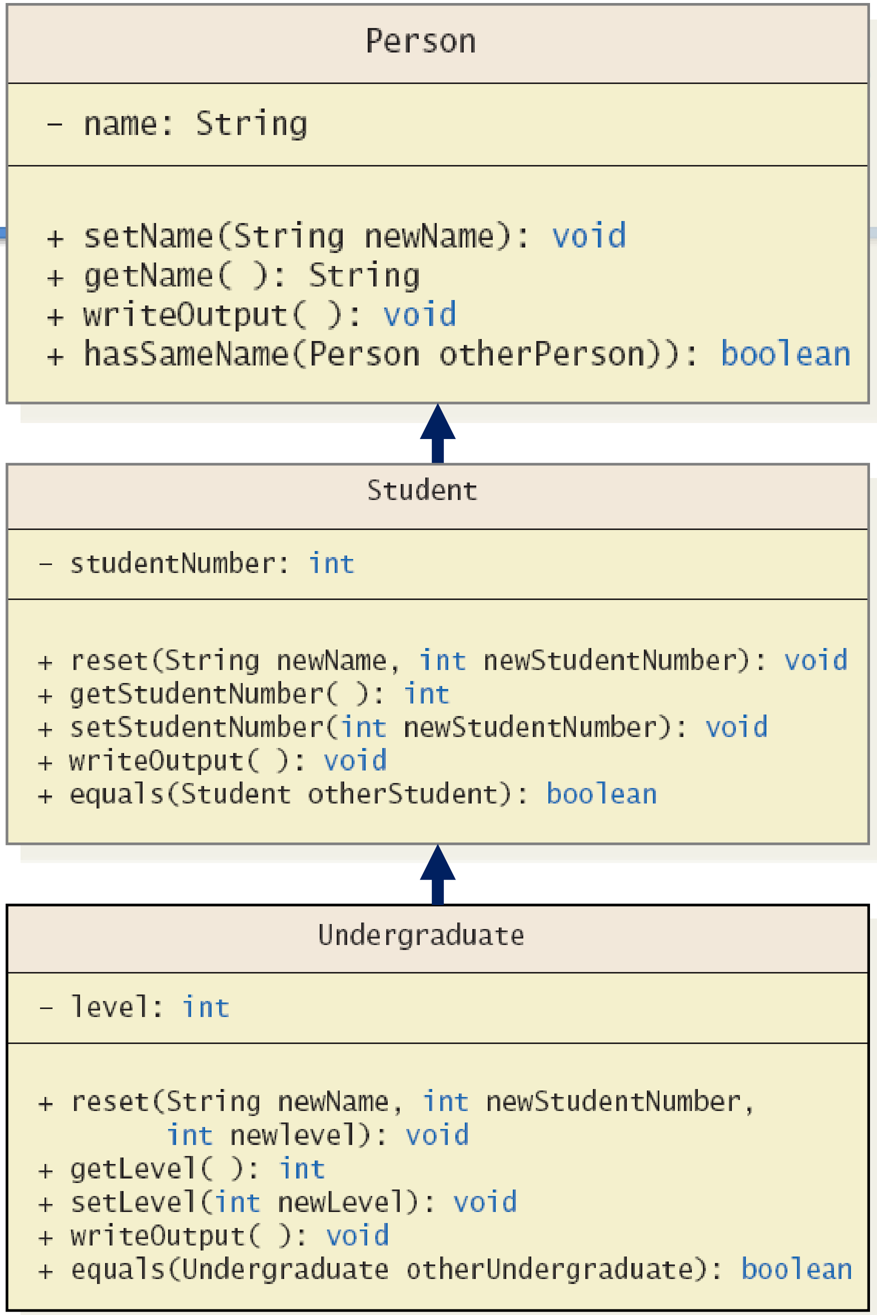
```
public class Undergradute extends Student
{
    private int level; //1 for freshman, 2 for sophomore
                       //3 for junior, or 4 for senior.
    public Undergraduate()
    {
        super();
        level = 1
    }
    public Undergraduate(String initialName,
                    int initialStudentNumber, int initialLevel)
    {
        super(initialName, initialStudentNumber);
        setLevel(initialLevel); //checks 1 <= initialLevel <= 4
    }
    public void reset(String newName, int newStudentNumber,
                      int newLevel)
    {
        reset(newName, newStudentNumber); //Student's reset
        setLevel(newLevel); //Checks 1 <= newLevel <= 4
    }

    public int getLevel()
    {
        return level;
    }
    public void setLevel(int newLevel)
    {
        if ((1 <= newLevel) && (newLevel <= 4))
            level = newLevel;
        else
        {
            System.out.println("Illegal level!");
            System.exit(0);
        }
    }
                                                        }
```



```
public void writeOutput()
{
    super.writeOutput();
    System.out.println("StudentLevel: " + level);
}
public boolean equals(Undergraduate otherUndergraduate)
{
    return equals(Student)otherUndergraduate) &&
           (this.level == otherUndergraduate.level);
}
```

Example

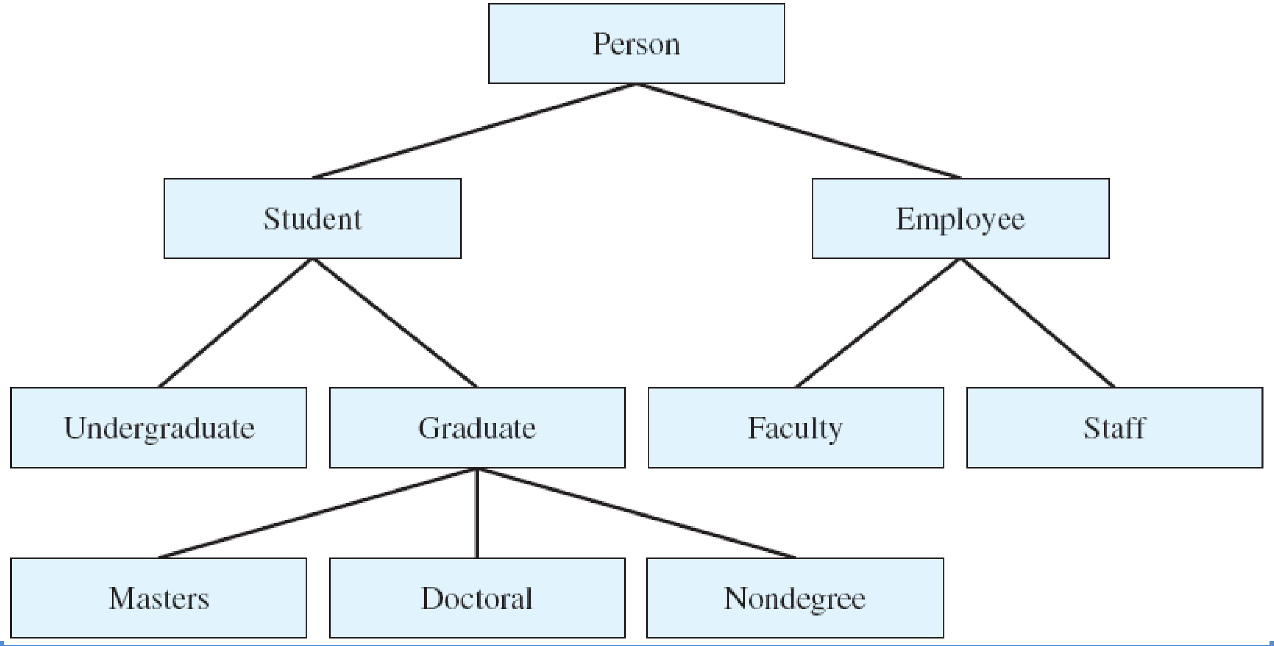
- Figure 8.4 More details of the UML class hierarchy



```
Person
- name: String
+ setName(String newName): void
+ getName( ): String
+ writeOutput( ): void
+ hasSameName(Person otherPerson)): boolean

Student
- studentNumber: int
+ reset(String newName, int newStudentNumber): void
+ getStudentNumber( ): int
+ setStudentNumber(int newStudentNumber): void
+ writeOutput( ): void
+ equals(Student otherStudent): boolean

Undergraduate
- level: int
+ reset(String newName, int newStudentNumber,
        int newlevel): void
+ getLevel( ): int
+ setLevel(int newLevel): void
+ writeOutput( ): void
+ equals(Undergraduate otherUndergraduate): boolean
```

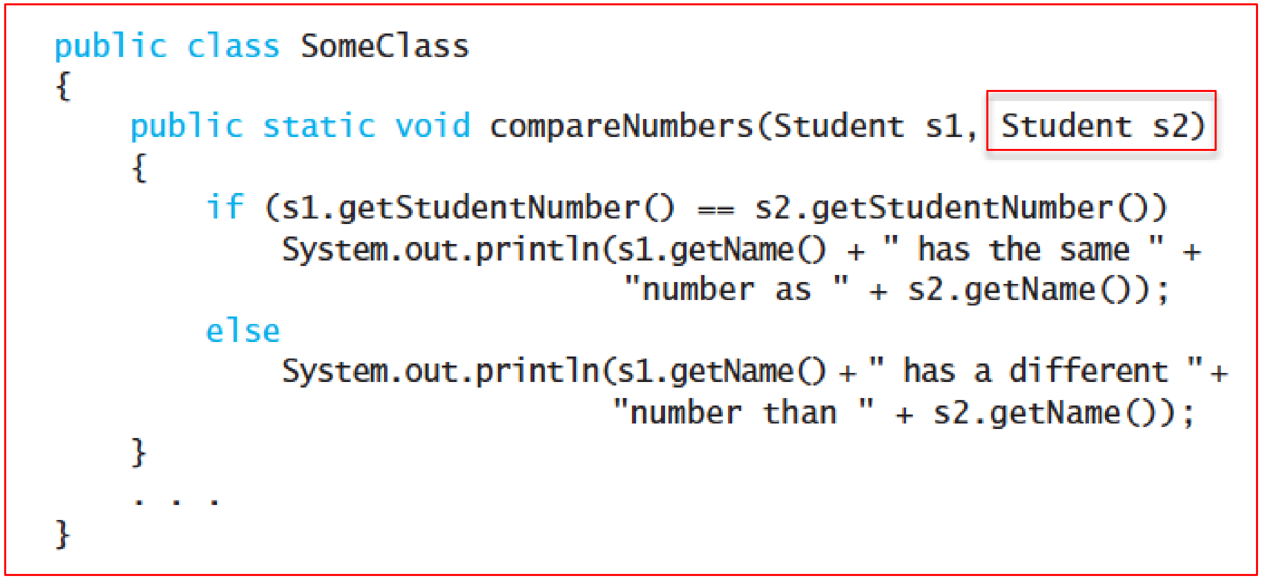
is-a Relationship

- Inheritance relationship is known as an ***is-a relationship***
  - An Undergraduate *is a* Student; A Student *is a* Person
  - A Student is an Undergraduate? Not necessarily!



Type compatibility

- Suppose:



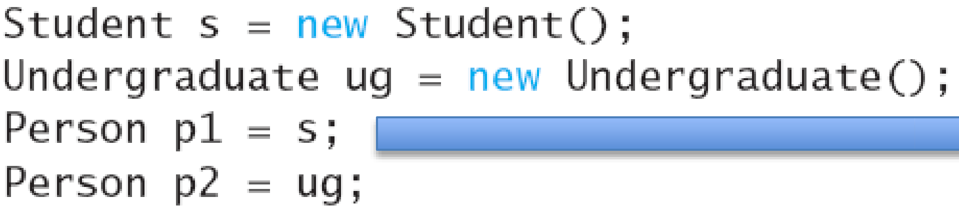
```
public class SomeClass
{
    public static void compareNumbers(Student s1, Student s2)
    {
        if (s1.getStudentNumber() == s2.getStudentNumber())
            System.out.println(s1.getName() + " has the same " +
                                "number as " + s2.getName());
        else
            System.out.println(s1.getName() + " has a different "+
                               "number than " + s2.getName());
    }
    . . .
}
```

- Does the following work?
  - Student s = new Student(“Mansoo”, 1234);
  - Undergraduate ug = new Undergraduate(“Jack”, 1234, 1);
  - SomeClass.compareNumbers(s, ug);

An object ug of the class Undergraduate is an object of Student ??

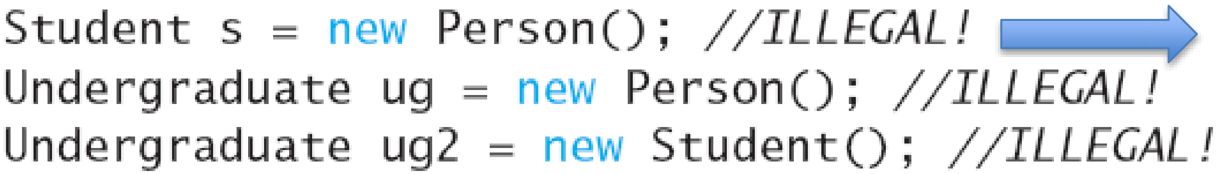
Type compatibility

- An object can have several types due to inheritance
  - An object of a class can be referenced by a variable of a base class
- An object of a derived class can behave as an object of the base class
  - E.g., **Every object of the class Undergraduate** is also an object of Student as well as Person
  - Note: this is not typecasting



```
Student s = new Student();
Undergraduate ug = new Undergraduate();
Person p1 = s;
Person p2 = ug;
```

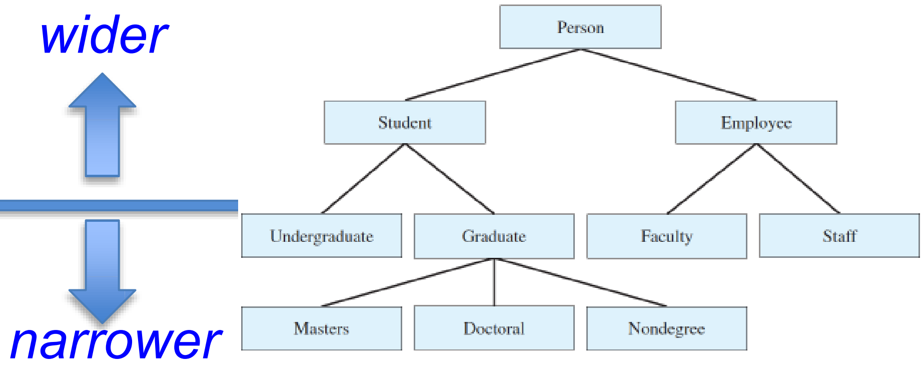
A Student *is a* Person
Ug is a Person



```
Student s = new Person(); //ILLEGAL!
Undergraduate ug = new Person(); //ILLEGAL!
Undergraduate ug2 = new Student(); //ILLEGAL!
```

Person is a student?
Person is a ug?

Type compatibility



- Basic rule
  - The classes higher up in the hierarchy are wider or general than those lower down
  - Similarly, lower classes are narrower or specific
- *Widening conversion*: assign a subtype to a supertype
  - *Fine!. it* can be checked at compile time. No action needed
- *Narrowing conversion:* convert a reference of a supertype into a reference of a subtype
  - must be explicitly converted by using the *cast* operator

Type compatibility

- Similar to primitive types, you can cast a variable to a different type
- Syntax Rule: `(Class_Name)variable_of_object`
  - *Person p = new Student();*
  - *Student s = (Student) p;*
- A run-time error happens if the cast is incorrect
  - *Person p = new Person();*
  - *Student s = (Student) p;* ***// WRONG! p cannot be cast to student***
  - *Doctoral d = (Doctoral) p;* ***// WRONG! p is not in Doctoral type***

Exercise

```
public class Car {
    public void run() { System.out.println("Car의 run 메소드"); }
}
public class Bus extends Car {
public void sound() {
    System.out.println("Bus 의 sound 메소드");
    }
    public void run() {
    System.out.println("Bus 의 run 메소드");
    }
    public static void main(String[] args) {
        Car c = new Bus(); // Can we change new Car?
        c.run();
        // c.sound(); // how can we use c.sound()?
        Bus b = c; // change here
        b.run();
        b.sound();
    }
}
```

Instanceof operator

- *instanceof*
  - You need to test an object's actual class using *instanceof* operator
  - It returns a boolean value indicating if an object is of a given class type
  - Similar to a comparison (==) operator

```
if (c instanceof Bus) {
    Bus b = (Bus) c;
    b.run();
    b.sound();
}
```

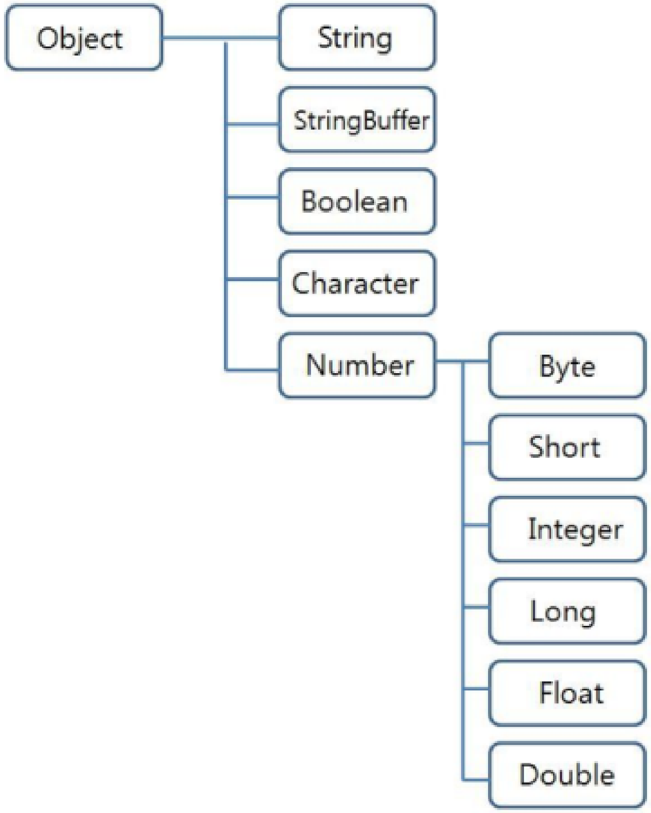
- Syntax:
  - if (*object* instanceof *ClassName*)
    ClassName newVar = (*ClassName*)*object*;
- Example:
  - if (obj instanceof String) String str = (String)obj;

Object Class

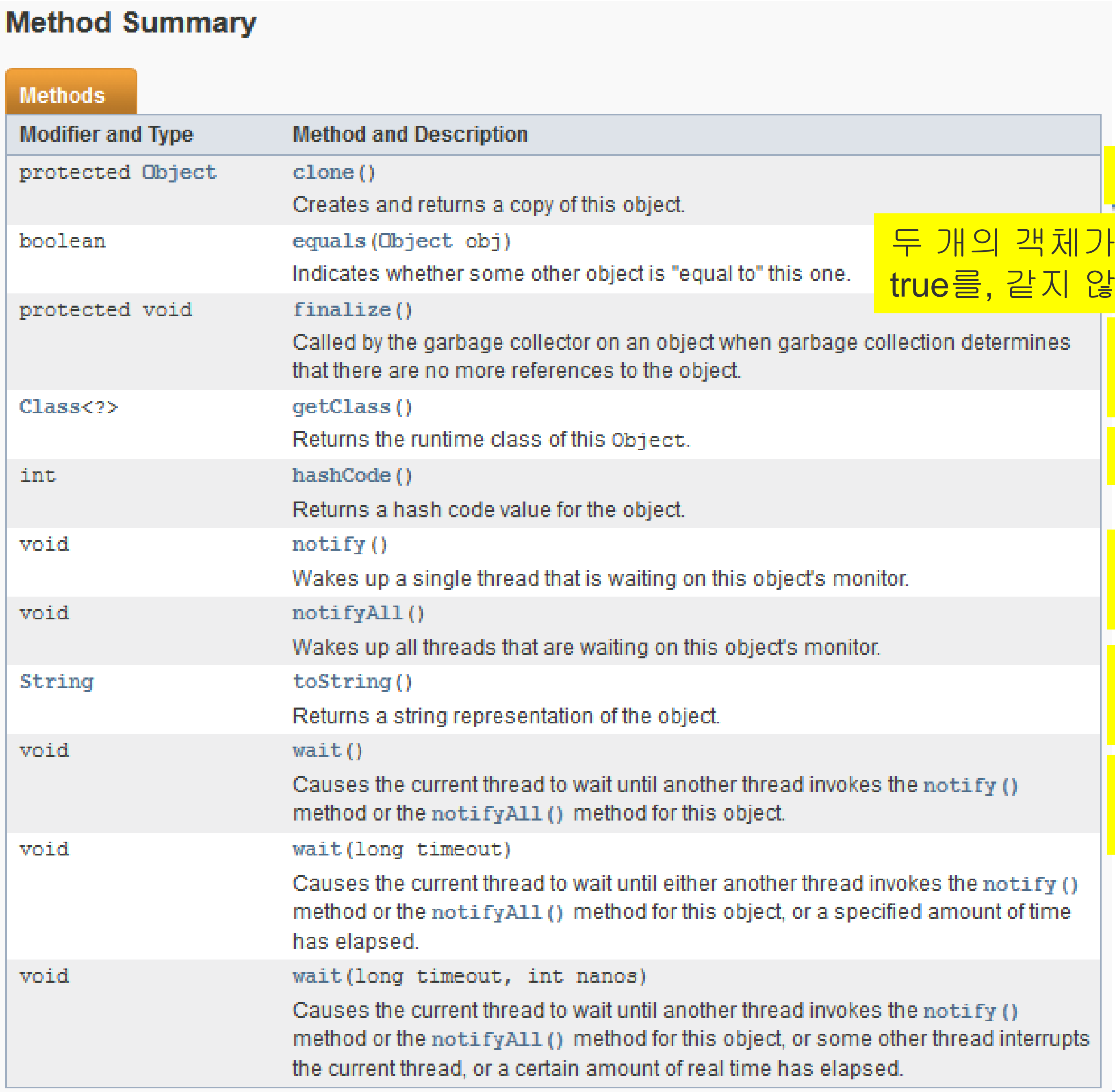
- Every class in Java inherits a base class “Object”
  - You don’t have to write “extends” explicitly
  - Every class in Java is an object
- Class Object has several methods and so every class inherits the public methods of object
  - Examples
    - Method equals
    - Method toString

e.g. java.lang package

No need to import!



https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html



Method Summary

Methods

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

현 객체를 복사

두 개의 객체가 같은지 비교하여 같으면 true를, 같지 않으면 false를 반환

가비지 컬렉션 직전에 객체의 리소스를 정리

객체의 클래스형을 반환

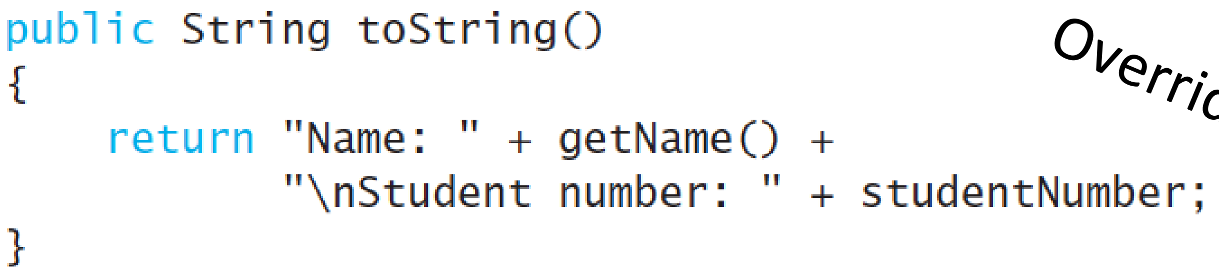
wait된 스레드 실행을 재개할 때 호출

현재 객체의 문자열을 반환

스레드를 일시적으로 중지

Object Class

- Example:
  Method toString called when println(theObject) invoked
  - Best to define your own toString to handle this

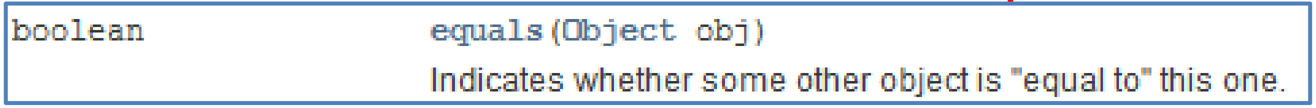


```
public String toString()
{
    return "Name: " + getName() +
           "\nStudent number: " + studentNumber;
}
```

Overriding!

A Better equals Method

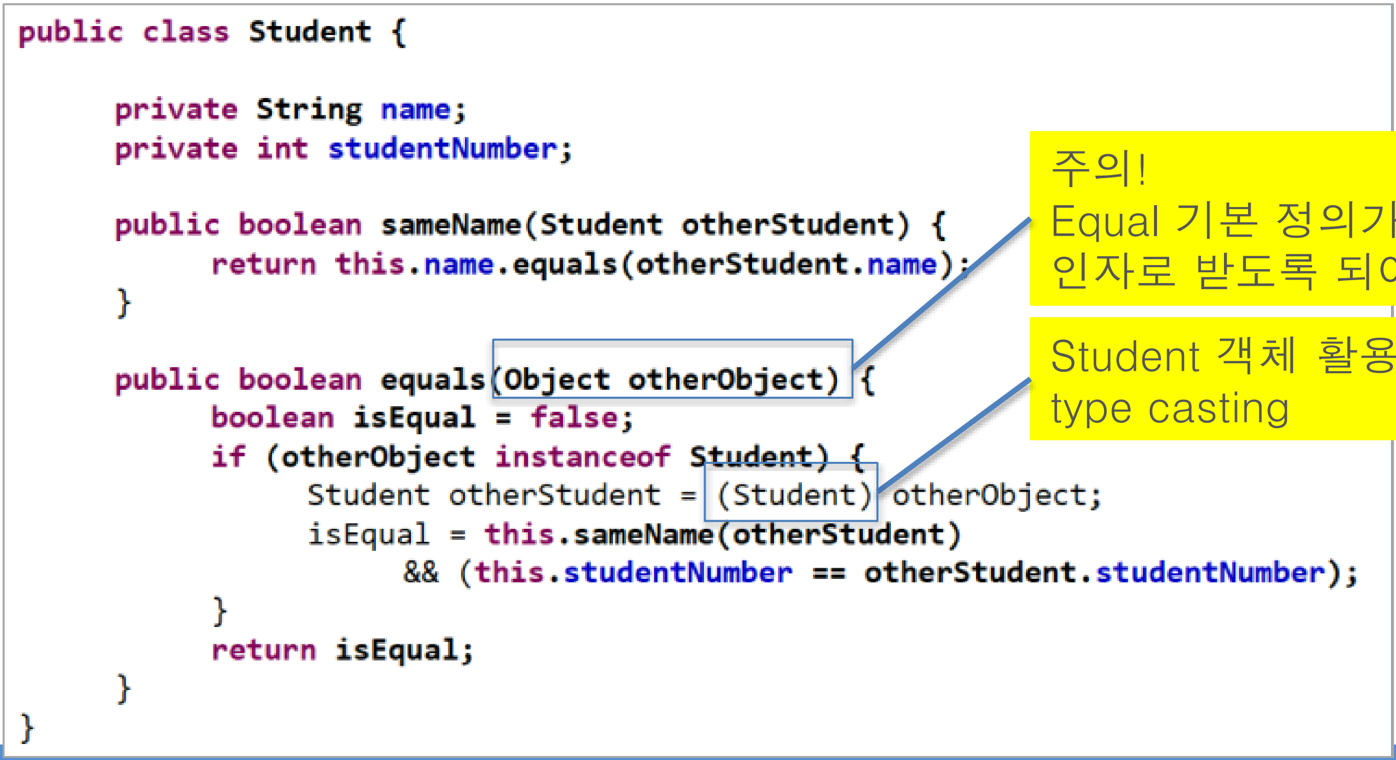
- Programmer of a class should override method equals from Object



```
boolean                equals(Object obj)
                       Indicates whether some other object is "equal to" this one.
```

- View code of sample override, listing 8.5

  public boolean equals(Object theObject)



```
public class Student {

    private String name;
    private int studentNumber;

    public boolean sameName(Student otherStudent) {
        return this.name.equals(otherStudent.name);
    }

    public boolean equals(Object otherObject) {
        boolean isEqual = false;
        if (otherObject instanceof Student) {
            Student otherStudent = (Student) otherObject;
            isEqual = this.sameName(otherStudent)
                && (this.studentNumber == otherStudent.studentNumber);
        }
        return isEqual;
    }
}
```

주의!
Equal 기본 정의가 object를 인자로 받도록 되어 있음

Student 객체 활용을 위해 type casting

Practice 6

- Ex6. Define a class Doctor extending Person
  - Attributes:
    - specialty (String): “Medicine”, “Surgery”, “Dentist”, or “Oriental”
    - visit_fee (double)
  - Methods
    - Constructors with 0 and 3 (name, specialty, fee) parameters: should call appropriate constructors of Person class
    - Accessor/mutator methods: check validity
    - toString(), equals(Object)
  - Write a test class
    - Run with ≥ 2 objects
    - Set two doctor instances
    - Compare two instances

Use this!

```java
public class Person {
    private String name;

    public Person() {
        name = "No name yet";
    }

    public Person(String initialName) {
        name = initialName;
    }

    public void setName(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }

    public void writeOutput() {
        System.out.println("Name:  " + name);
    }

    public boolean hasSameName(Person otherPerson) {
        return this.name.equalsIgnoreCase(otherPerson.name);
    }
}
```

Fill out the doctor class
and write a test class (or method)

```java
public class Doctor extends Person {
    String specialty;
    double visit_fee;

    public Doctor() { }
    public Doctor(String name, String specialty, double visit_fee) { }

    public void setSpecialty(String specialty) {
        String major[] = { "Medicine", "Surgery", "Dentist", "Oriental" };
       // define your code more!
    }
    public String getSpecialty() { }
    public void setVisitFee(double visit_fee) { }
    public double getVisitFee() { }

    @Override
    public String toString() { }

    @Override
    public boolean equals(Object otherObject) { }
}
```

8.3 Polymorphism

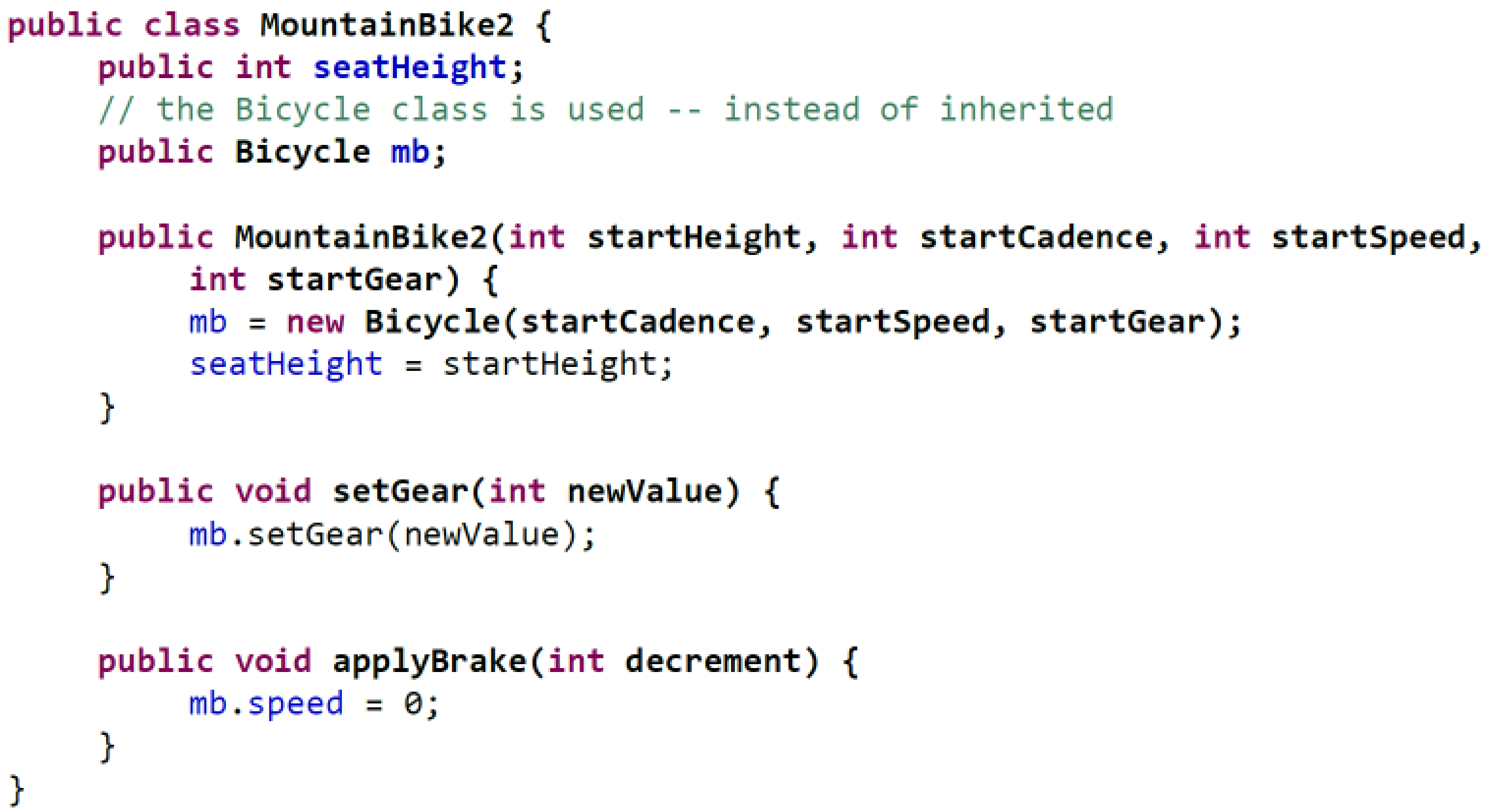
Question…

Why do we use inheritance ?

To reuse codes??

Let’s see

- You can reuse codes without inheritance!!



```
public class MountainBike2 {
    public int seatHeight;
    // the Bicycle class is used -- instead of inherited
    public Bicycle mb;

    public MountainBike2(int startHeight, int startCadence, int startSpeed,
        int startGear) {
        mb = new Bicycle(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    public void setGear(int newValue) {
        mb.setGear(newValue);
    }

    public void applyBrake(int decrement) {
        mb.speed = 0;
    }
}
```

Inheritance is not all for reusability

- **Inheritance can be good for reusability**
- But, it is **not intended** for reusability
  - That means, if you want to reuse your code, you shall **not** think about inheritance first!

- **Inheritance is for flexibility !!**
  - It is used when different objects need different methods
  - We call this property **"polymorphism"**
    - We will see soon.

Polymorphism

- It means “many forms”
- Same instruction to mean different things in different contexts.
  - Example: “Go play your favorite sport.”
    - I’d go play soccer
    - Others of you would play basketball or football instead.
- In programming, this means that the **same** method name can cause **different actions** depending on what object it is applied to

Why is Polymorphism Required?

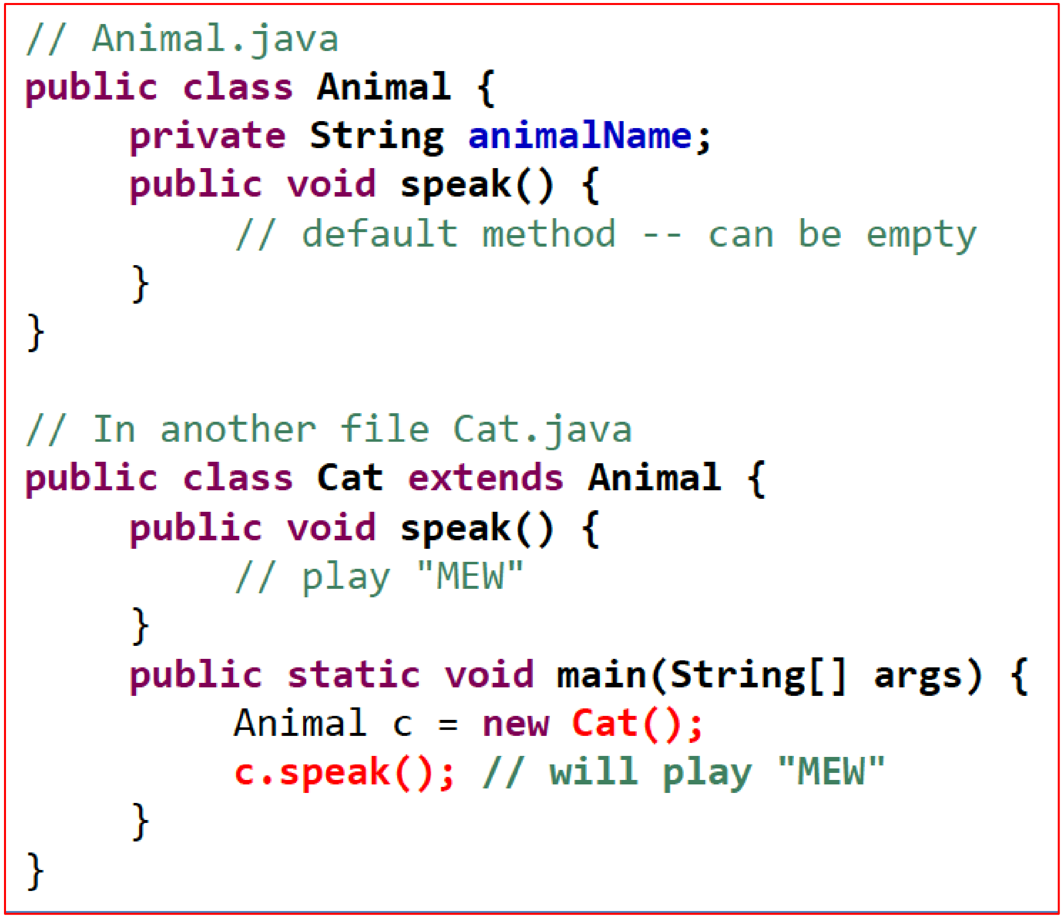
- Let's consider if we want to design a set of classes that represents animals
  - Every animal can play its own sound

- How to write a method for playing sound ?

Animal Class without Polymorphism

```
public class Animal {
    private String animalName;
    private String species;
    private void playDuckSound() {
        // play "QUACK"
    }
    private void playDogSound() {
        // play "WOOF"
    }
        private void playCatSound() {
    // play "MEW"
    }
    public void speak() {
        if (species.equals("Duck")) {
            this.playDuckSound();
        } else if (species.equals("Dog")) {
            this.playDogSound();
        } else if (species.equals("Cat")) {
            this.playCatSound();
        }
    }
}
```

Polymorphism and Overriding



```
// Animal.java
public class Animal {
     private String animalName;
     public void speak() {
          // default method -- can be empty
     }
}

// In another file Cat.java
public class Cat extends Animal {
     public void speak() {
          // play "MEW"
     }
     public static void main(String[] args) {
          Animal c = new Cat();
          c.speak(); // will play "MEW"
     }
}
```

- Key Point:

When you invoke the methods from the superclass variable, the overridden method is called

VERY IMPORTANT!

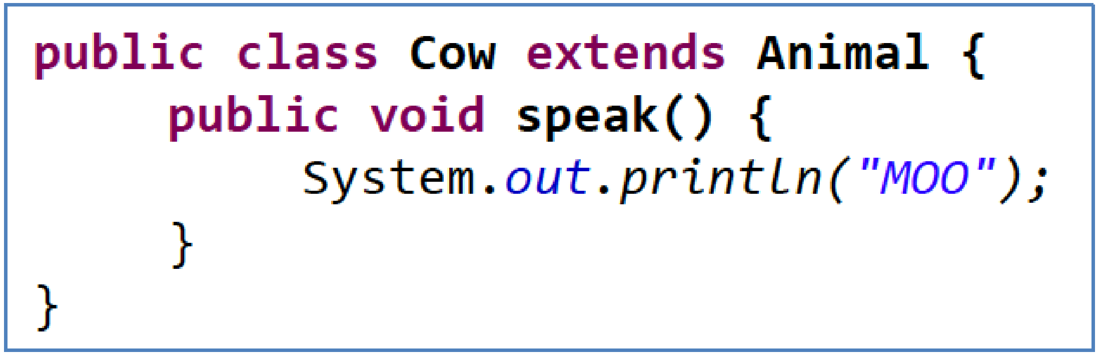
```java
public class Animal {
    private String animalName;
    public void speak() {
    // default method -- can be empty
    }

    public static void main(String[] args)
    {
        Animal a[] = new Animal[3];
        a[0] = new Cat();
        a[1] = new Dog();
        a[2] = new Duck();
        for (int i = 0; i < 3; i++) {
            a[i].speak();
        }
    }
}
```

```java
public class Cat extends Animal {
    public void speak() {
        System.out.println("MEW");
    }
}

public class Dog extends Animal {
    public void speak() {
        System.out.println("WOOF");
    }
}

public class Duck extends Animal {
    public void speak() {
        System.out.println("QUACK");
    }
}
```

OUTPUT = ?

Polymorphism

- What if we want to add a new animal: cow?
  - Just write a new class Cow
    - Nothing in Animal shall be changed



```java
public class Cow extends Animal {
    public void speak() {
        System.out.println("MOO");
    }
}
```

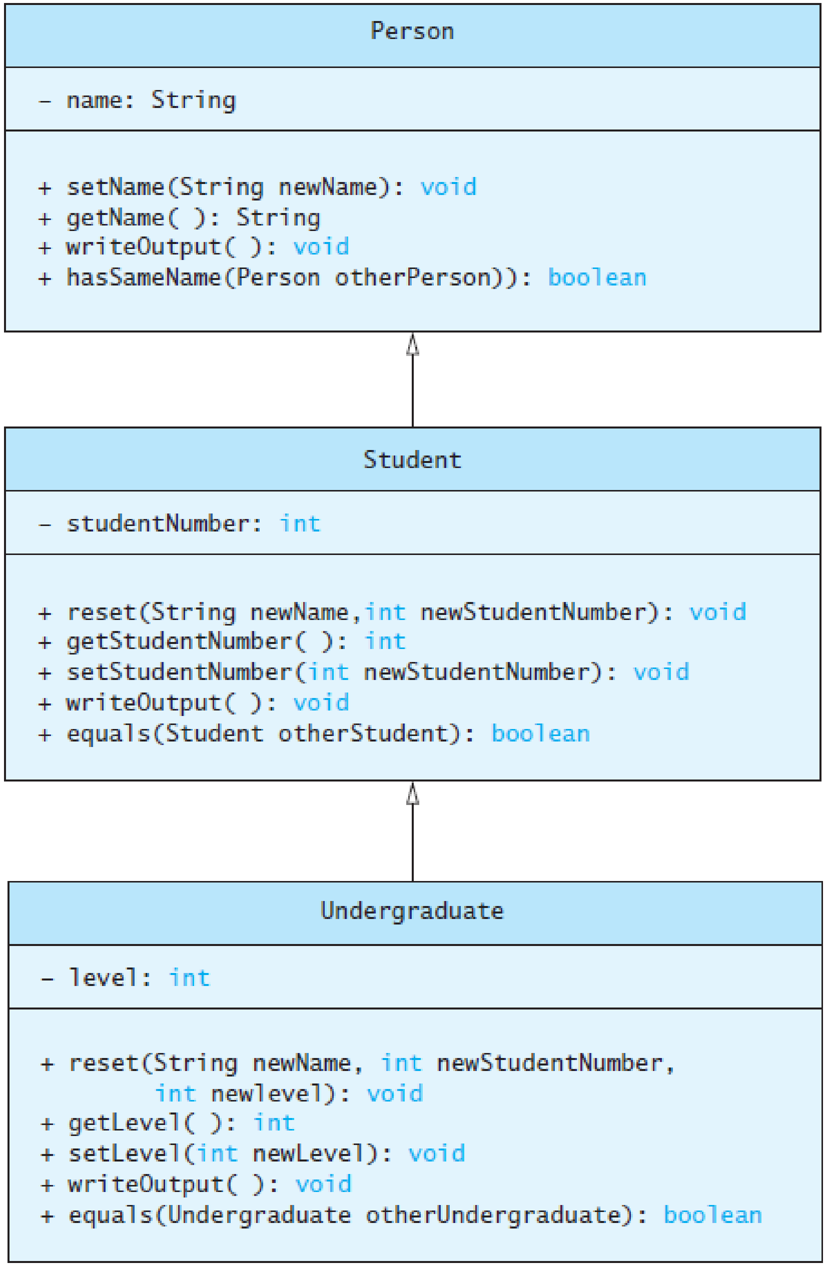
Polymorphism

- 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);
```



Person

- name: String

+ setName(String newName): void
+ getName( ): String
+ writeOutput( ): void
+ hasSameName(Person otherPerson)): boolean

Student

- studentNumber: int

+ reset(String newName,int newStudentNumber): void
+ getStudentNumber( ): int
+ setStudentNumber(int newStudentNumber): void
+ writeOutput( ): void
+ equals(Student otherStudent): boolean

Undergraduate

- level: int

+ reset(String newName, int newStudentNumber, int newlevel): void
+ getLevel( ): int
+ setLevel(int newLevel): void
+ writeOutput( ): void
+ equals(Undergraduate otherUndergraduate): boolean

Polymorphism

- Given:

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

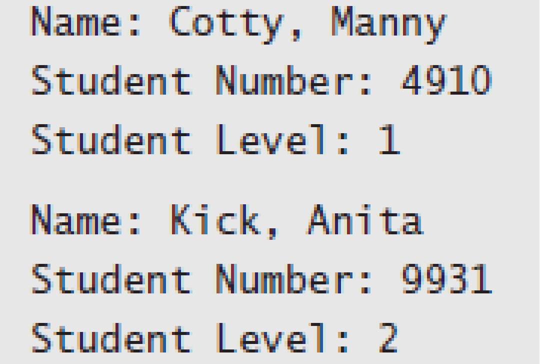
- When invoking:

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

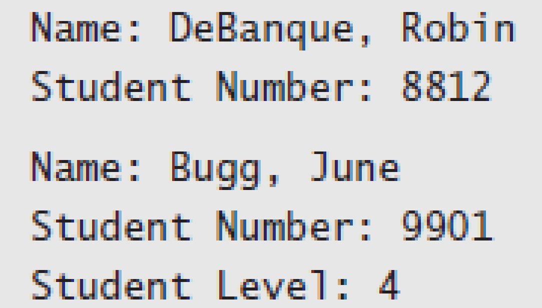
- Which `writeOutput()` is invoked, the one defined for `Student` or the one defined for `Person`?
- Answer: The one defined for `Student`

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
```

```
public class Person{
    private String name;
    public Person ()    {       name = "No name yet";    }
    public Person (String initialName)    {       name = initialName;    }
    public void setName (String newName)    {       name = newName;    }
    public String getName ()    { return name;    }
    public void writeOutput ()    {  System.out.println ("Name:  " + name);    }
    public boolean hasSameName (Person otherPerson) {  return this.name.equalsIgnoreCase
otherPerson.name);
    }
public class Student extends Person{
    private int studentNumber;
    public Student ()    {
        super ();
        studentNumber = 0; //Indicating no number yet
    }
    public Student (String initialName, int initialStudentNumber)    {
        super (initialName);
        studentNumber = initialStudentNumber;
    }
    public void reset (String newName, int newStudentNumber)    {
        setName (newName);
        studentNumber = newStudentNumber;
    }
    public int getStudentNumber ()    {       return studentNumber;    }
    public void setStudentNumber (int newStudentNumber)    {       studentNumber = newStudentNumber;    }
    public void writeOutput ()    {
        System.out.println ("Name:  " + getName ());
        System.out.println ("Student Number:  " + studentNumber);
    }
    public boolean equals (Student otherStudent)    {
        return this.hasSameName (otherStudent) &&
            (this.studentNumber == otherStudent.studentNumber);
    }
}
```

```java
public class Undergraduate extends Student{
    private int level; //1 for freshman, 2 for sophomore,
    //3 for junior, or 4 for senior.
     public Undergraduate ()    {
        super ();
        level = 1;
    }
    public Undergraduate (String initialName, int initialStudentNumber, int initialLevel)    {
        super (initialName, initialStudentNumber);
        setLevel (initialLevel); //Checks 1 <= initialLevel <= 4
    }
    public void reset (String newName, int newStudentNumber, int newLevel)    {
        reset (newName, newStudentNumber); //Students reset
        setLevel (newLevel); //Checks 1 <= newLevel <= 4
    }
    public int getLevel ()    {        return level;    }
    public void setLevel (int newLevel)    {
        if ((1 <= newLevel) && (newLevel <= 4))            level = newLevel;
        else        {
            System.out.println ("Illegal level!");
            System.exit (0);
        }
    }
    public void writeOutput ()    {
        super.writeOutput ();
        System.out.println ("Student Level: " + level);
    }
    public boolean equals (Undergraduate otherUndergraduate)    {
        return equals ((Student) otherUndergraduate) &&
            (this.level == otherUndergraduate.level);
    }
}
```

Test class

```
public class PolymorphismDemo {
    public static void main(String[] args){
        Person[] people = new Person[4];

        people[0] = new Undergraduate("Cotty, Manny", 4910, 1);
        people[1] = new Undergraduate("Kick, Anita", 9931, 2);
        people[2] = new Student("DeBanque, Robin", 8812);
        people[3] = new Undergraduate("Bugg, June", 9901, 4);

        for (Person p : people)
        {
        p.writeOutput();
        System.out.println();
        }
    }
}
```

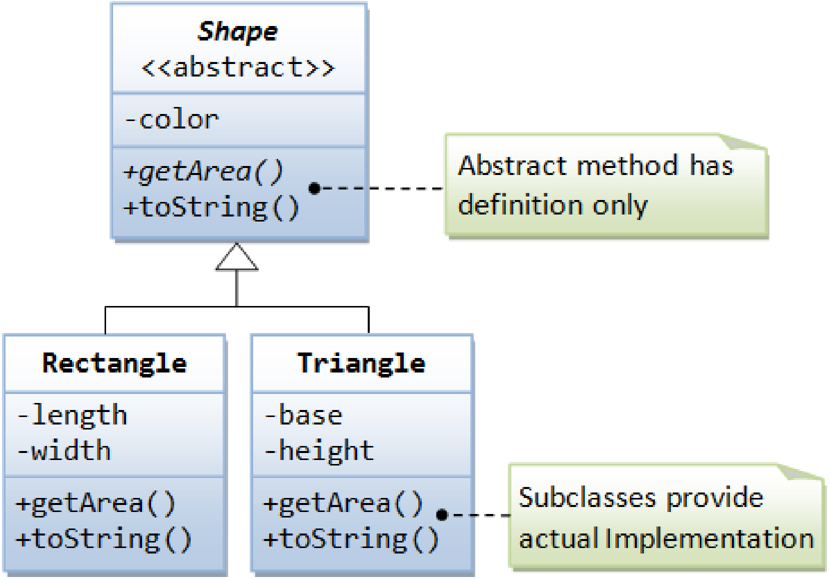
8.4 Interfaces and Abstract Classes

abstract Classes

- Abstract method
  - A method with only signature (method name, a list of arguments, and return type)
  - No implementation (method body)
  - Use the keyword abstract for its declaration
- Abstract class
  - A class containing one or more abstract methods
  - Abstract classes cannot have an instance
  - An abstract class must be declared with a class-modifier abstract

abstract method

- For example, in the Shape class, we can declare the abstract methods getArea() as follows:



Abstract method has definition only

Subclasses provide actual Implementation

```
abstract public class Shape {
   // Private member variable
   private String color;
   // Constructor
   public Shape (String color) {
      this.color = color;
   }
   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }
 // All Shape subclasses must implement a method
called getArea()
   abstract public double getArea();
}
```

Why using abstract methods?

- Implementation of these methods is not possible in the base class, since the actual implementation will be defined in subclasses (not yet known)
  - Example:
    - The method getArea() in the Shape class is not yet known! (How to compute the area if the shape is not known?)
  - Implementation of these abstract methods will be provided later once the actual shape is known.
- These abstract methods cannot be invoked because they have no implementation.
  - E.g., **Shape s = new Shape();** **~~s.getArea();~~**

abstract Classes

- An abstract class must be declared with a class-modifier **abstract**.
  - A class containing **one or more** **abstract methods** is called an abstract class.

```
abstract public class Shape {
   // Private member variable
   private String color;

   // Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }
   // All Shape subclasses must implement a method called getArea()
    abstract public double getArea();
}
```

abstract Classes

- Not all methods of an abstract class are abstract methods
- Abstract class makes it easier to define a base class
  - Specifies the obligation of designer to override the abstract methods for each subclass

How to use abstract class

- Accessing by **inheritance!**

```
public class abstractTest {

public static void main(String[] args) {
        A obj1 = new A(); //error!
        B obj2 = new B();
    }
}
abstract class A{
    public abstract int b();
    public void c(){ System.out.println("error");}
    public void d(){ System.out.println("test"); }
}
class B extends A{
    public int b(){ return 1;}
}
```

Lab: calculator

```
abstract class Calculator{
    int left, right;
    public void setOprands(int left, int right){
        this.left = left;
        this.right = right;
    }
    public abstract void sum();
    public abstract void avg();
    public void run(){
        sum();
        avg();
    }
}
```

산술 평균

```
class CalculatorA extends Calculator {
    public void sum(){
        System.out.println("+ sum :"+(this.left+this.right));
    }
    public void avg(){
        System.out.println("+ avg :"+(this.left+this.right)/2);
    }
}
```

조화 평균

```
class CalculatorB extends Calculator {
    public void sum(){
        System.out.println("- sum :"+(this.left+this.right));
    }
    public void avg(){
        System.out.println("- avg :"+(2/(1/this.left)+(1/this.right));
    }
}
```

```
public class CalculatorDemo {
    public static void main(String[] args) {
        CalculatorA c1 = new CalculatorA();
        c1.setOprands(10, 20);
        c1.run();

        CalculatorB c2 = new CalculatorB();
        c2.setOprands(10, 20);
        c2.run();
    }

}
```

Interfaces

- A way to describe **what classes should do**, without specifying how they should do it
- Contains headings for a number of public methods
  - All methods are public abstract methods
- A set of requirements for a class that wants to conform to the interface
- Example:

```java
public interface Measurable {
    public static final int INCHES_PER_FOOT = 12;
    // Returns the perimeter
    public double getPerimeter();
    // Returns the area
    public double getArea();
}
```

Interface declarations

- Interface members
  - Constants (fields)
  - Method signatures
  - Nested classes and interfaces
- Does not include:
  - Declarations of constructors
  - Instance variables
  - Method bodies
- Interface name begins with uppercase letter
- Stored in a file with suffix .java

Make a Class Implementing an Interface

- **Two steps to make a class implement an interface**

  1. declare that the class intends to implement the given interface by using the `implements` keyword

     ```
     implements Interface_name
     ```

     e.g., `class Employee implements Comparable { . . . }`

  2. Define all specified methods in the interface

Example: Rectangle class

```
/** A class of rectangles. */
public class Rectangle implements Measurable{
    private double myWidth;
    private double myHeight;

    public Rectangle (double width, double height)
    {
        myWidth = width;
        myHeight = height;
    }

    public double getPerimeter()
    {
        return 2 * (myWidth + myHeight);
    }

    public double getArea()
    {
        return myWidth * myHeight;
    }
}
```

```
public interface Measurable {
    public static final int
INCHES_PER_FOOT = 12;
    // Returns the perimeter
    public double getPerimeter();
    // Returns the area
    public double getArea();
}
```

Example: Circle class

```
/** A class of circles. */
public class Circle implements Measurable{
    private double myRadius;

    public Circle(double radius)
    {
        myRadius = radius;
    }

    public double getPerimeter ()
    {
        return 2 * Math.PI * myRadius;
    }

    public double getCircumference ()
    {
        return getPerimeter ();
    }

    public double getArea ()
    {
        return Math.PI * myRadius * myRadius;
    }
}
```

```
public interface Measurable {
    public static final int
INCHES_PER_FOOT = 12;
    // Returns the perimeter
    public double getPerimeter();
    // Returns the area
    public double getArea();
}
```

Interface as a type

- Interfaces are not classes
  - Measurable x = new Measurable(); // WRONG!
- You can still declare an interface variable; it refers to an object of a class that implements the interface
  - Measurable m = new Rectangle(); // OK
- Benefits?
  - Allows you to view classes that are not related at all to a single type.

```
Measurable[] arr = new Shape[2];
   arr[0] = new Rectangle( );
   arr[1] = new Circle( );
```

Extending interfaces

- Interfaces support multiple inheritance
  - **An interface can extend more than one interface**
  - Superinterfaces and subinterfaces
- A class that implements the new interface must implement all the methods of both interfaces

Example

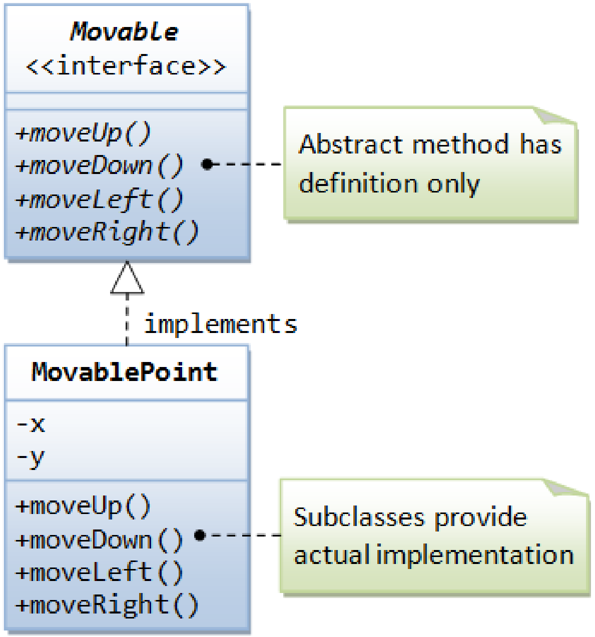
```
 public interface SerializableRunnable extends
java.io.Serializable, Runnable {
     . . .
 }
```

Why using interfaces?

- An interface is a *contract* (or a protocol-규약, or a common understanding) of what the classes can do.
  - When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface.
- 1. interfaces provide **a *communication contract* between two objects.**
  - If you know a class implements an interface, then you are guaranteed to be able to invoke these methods safely
- 2. Java does not support multiple inheritance; that is supplemented by “multiple implementation of interfaces”

Lab: Movable Interface

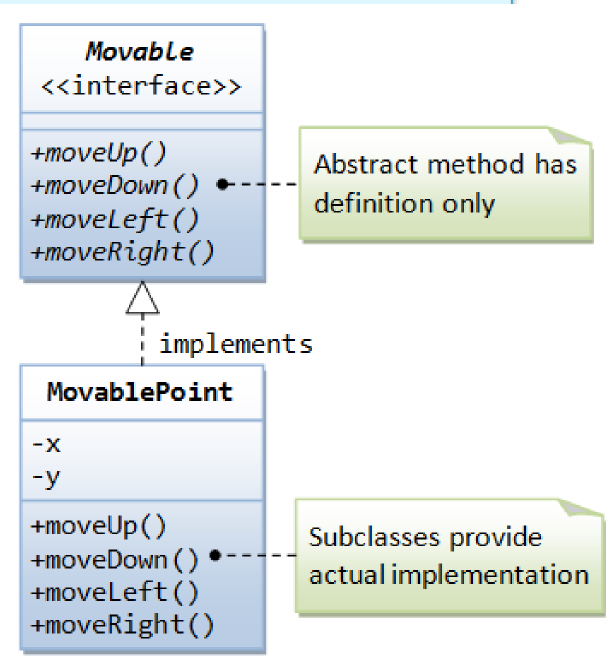
- Suppose that our application involves many objects that can move. We could define an interface called ***Movable***, containing the signatures of the various movement method.



1. Define interface *Movable*!

- Moveable.java

```
public interface Movable {
   // abstract methods to be implemented by the subclasses

}
```



2. MovablePoint.java

```
public class MovablePoint implements Movable
{
   // Private member variables
   private int x, y;    // (x, y) coordinates
of the point

   // Constructor
   public MovablePoint(int x, int y) {

   }

   @Override
   public String toString() {

")
   }
```

```
// Implement abstract methods
// defined in the interface Movable
   @Override
   public void moveUp() {


   @Override
   public void moveDown() {


   @Override
   public void moveLeft() {


   @Override
   public void moveRight() {


}
```

Test Program

```
public class TestMovable {
   public static void main(String[] args) {
      Movable m1 = new MovablePoint(5, 5);  // upcast
      System.out.println(m1);    // (5,5)
      m1.moveDown();
      System.out.println(m1);    // (5,6)
      m1.moveRight();
      System.out.println(m1);    // (6,6)
   }
}
```

Lab: 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

Sorting an Array of Fruit Objects

```java
public class Fruit {
    private String fruitName;

    public Fruit()    {
        fruitName = "";
    }
    public Fruit(String name)   {
        fruitName = name;
    }
    public void setName(String name) {
        fruitName = name;
    }
    public String getName() {
        return fruitName;
    }
}
```

```java
import java.util.Arrays;
public class FruitDemo {
    public static void main(String[] args) {
        Fruit[] fruits = new Fruit[4];

        fruits[0] = new Fruit("Orange");
        fruits[1] = new Fruit("Apple");
        fruits[2] = new Fruit("Kiwi");
        fruits[3] = new Fruit("Durian");

        Arrays.sort(fruits);

        // Output the sorted array of fruits
        for (Fruit f : fruits)        {
        System.out.println(f.getName());
        }
    }
}
```

```
Exception in thread "main" java.lang.ClassCastException: Fruit cannot be
cast to java.lang.Comparable
at java.util.ComparableTimSort.countRunAndMakeAscending(Unknown Source)
at java.util.ComparableTimSort.sort(Unknown Source)
at java.util.Arrays.sort(Unknown Source)
at FruitDemo.main(FruitDemo.java:14)
```

Comparable interface

- Why errors in Array.sort()?

```
Exception in thread "main"
java.lang.ClassCastException: Fruit
cannot be cast to
java.lang.Comparable
```

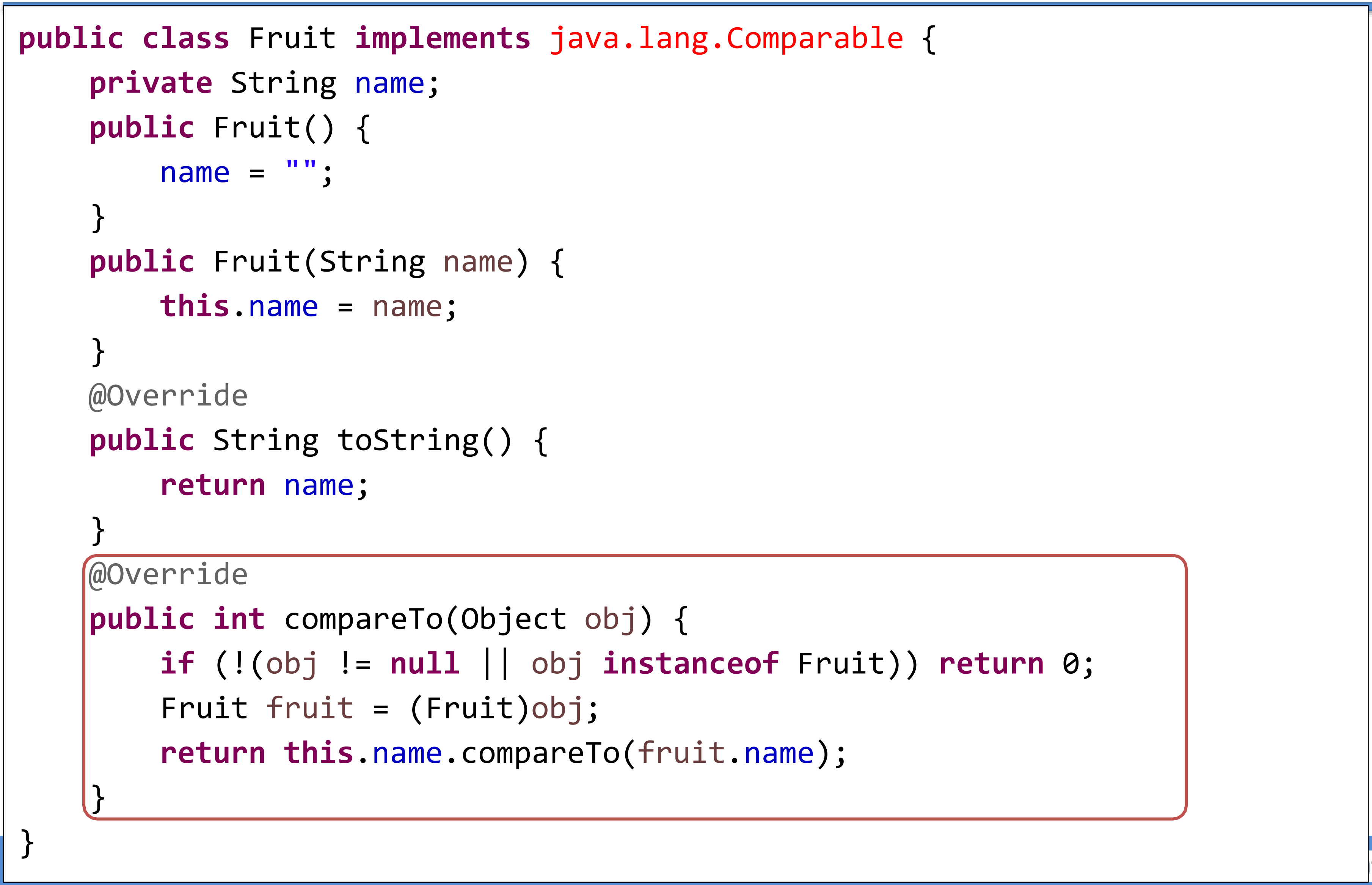
- As soon as sort () is executed, it is sorted internally according to **the content of the method through compareTo ()**.
- Comparable interface
  - A predefined interface in Java
  - Impose an ordering upon objects that implement it
  - Requires *compareTo*() method to be implemented

Comparable interface

```
public interface Comparable
       {
          int compareTo(Object otherObject);
       }
```

This requires that any class implementing the Comparable interface contains a compareTo method, and this method must take an Object parameter and return an integer

Comparable interface



```java
public class Fruit implements java.lang.Comparable {
    private String name;
    public Fruit() {
        name = "";
    }
    public Fruit(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return name;
    }
    @Override
    public int compareTo(Object obj) {
        if (!(obj != null || obj instanceof Fruit)) return 0;
        Fruit fruit = (Fruit)obj;
        return this.name.compareTo(fruit.name);
    }
}
```

compareTo Method

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

```
public class Fruit implements Comparable{   …
    public int compareTo(Object o)   {
       if ((o != null) && (o instanceof Fruit))    {
           Fruit otherFruit = (Fruit) o;
    /* Alternate definition of comparison using fruit length */
        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
    }
```

Interfaces and abstract classes

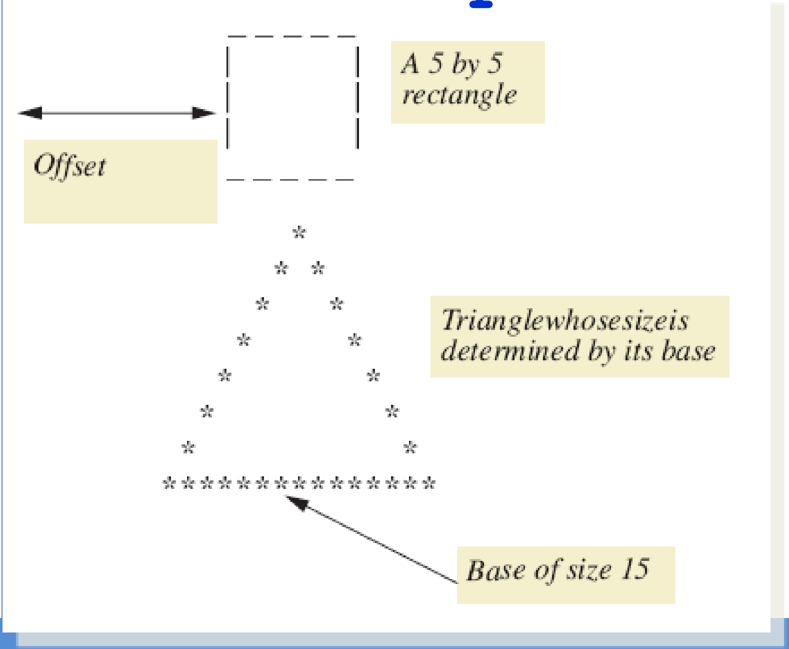
- Why bother introducing two concepts: abstract class and interface?

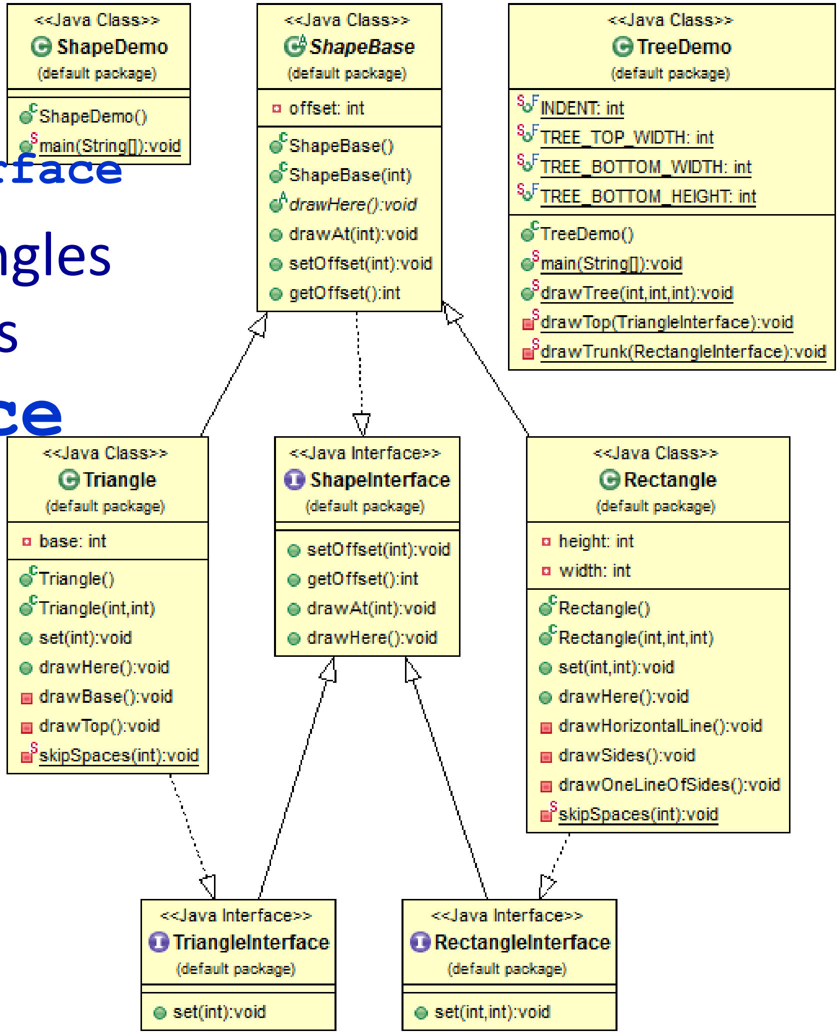
```
abstract class Comparable  {
  public abstract int compareTo (Object otherObject);
}
class Employee extends Comparable  {
    pulibc int compareTo(Object otherObject) { . . . }
}
-----------------------------------------------------
public interface Comparable {
    int compareTo (Object otherObject)
}
class Employee implements Comparable  {
    public int compareTo (Object otherObject) { . . . }
}
```

- A class can only extend a single abstract class, but it can implement as many interfaces as it wants
- An abstract class can have a partial implementation, protected parts, static methods and so on, while interfaces are limited to public constants and public methods with no implementation

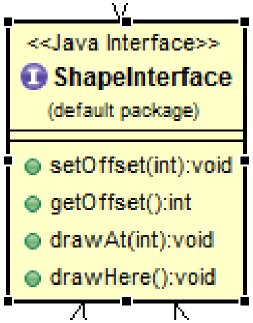
Lab: Character Graphics

- View interface for simple shapes, listing 8.10 `interface ShapeInterface`
- To create classes that draw rectangles and triangles, we create interfaces that extend `ShapeInterface`





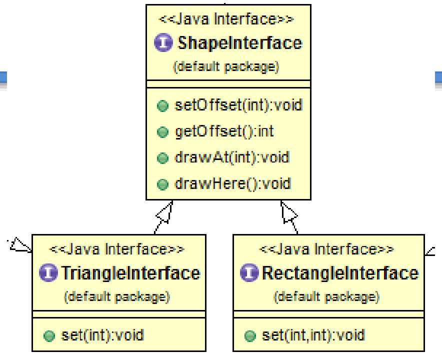
- All member variables must be **public static final** and can be omitted.
- All methods must be **public abstract** and can be omitted.



<<Java Interface>>
ShapeInterface
(default package)

setOffset(int):void
getOffset():int
drawAt(int):void
drawHere():void

```
/*Interface for simple shapes drawn on
the screen using keyboard characters.*/
public interface ShapeInterface
{
    /*Sets the offset for the shape.  */
    public void setOffset (int newOffset);
    /*Returns the offset for the shape.*/
    public int getOffset ();

    /* Draws the shape at lineNumber lines
      down from the current line.    */
    public void drawAt (int lineNumber);
    /* Draws the shape at the current line.
    */
    public void drawHere ();
}
```



<<Java Interface>>
ShapeInterface
(default package)

setOffset(int):void
getOffset():int
drawAt(int):void
drawHere():void

<<Java Interface>>
TriangleInterface
(default package)

set(int):void

<<Java Interface>>
RectangleInterface
(default package)

set(int,int):void

```
/*Interface for a rectangle to be drawn on
the screen.*/
public interface RectangleInterface extends
ShapeInterface
{
    /* Sets the rectangle's dimensions.  */
    public void set (int newHeight, int
newWidth);
}

/*Interface for a triangle to be drawn on
the screen.*/
public interface TriangleInterface extends
ShapeInterface{
    /*    Sets the triangle's base.  */
    public void set (int newBase);
}
```

Lab: Character Graphics

- 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`**

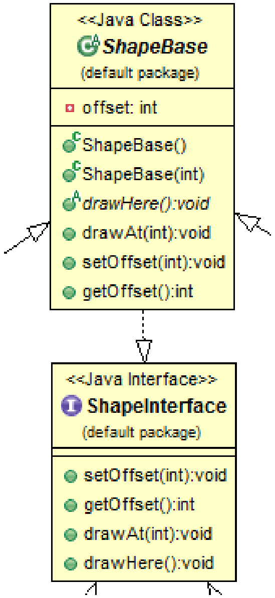
```
public class ShapeBasics implements ShapeInterface {
    private int offset;
    public ShapeBasics ()    {
        offset = 0;
    }
    public ShapeBasics (int theOffset)    {
        offset = theOffset;
    }
    public void setOffset (int newOffset)    {
        offset = newOffset;
    }
    public int getOffset ()    {
        return offset;
    }
```

위에서 몇칸을 띄워서 그릴 것인가. 결국 그림은 drawHere에서 그림

```
    public void drawAt (int lineNumber)    {
        for (int count = 0 ; count < lineNumber ; count++)
            System.out.println ();
        drawHere ();
    }
```

옆으로 몇칸을 띄워서 그릴것인가

```
    public void drawHere () {
        for (int count = 0 ; count < offset ; count++)
            System.out.print (' ');
        System.out.println ('*');
    }
}
```

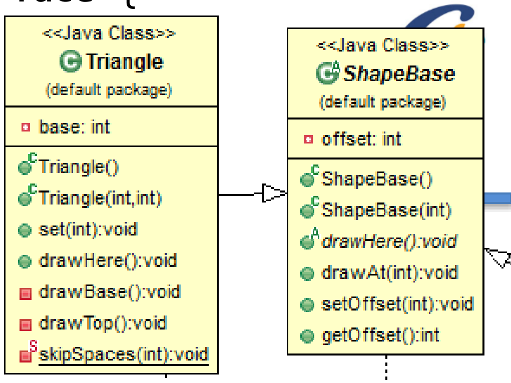


```
/*Interface for simple shapes drawn on
the screen using keyboard characters.*/
public interface ShapeInterface
{
    /*Sets the offset for the shape. */
    public void setOffset (int newOffset);
    /*Returns the offset for the shape.*/
    public int getOffset ();

    /* Draws the shape at lineNumber lines
      down from the current line.    */
    public void drawAt (int lineNumber);
    /* Draws the shape at the current line.
    */
    public void drawHere ();
}
```

```
public class Rectangle extends ShapeBasics implements RectangleInterface {
    private int height;
    private int width;
    public Rectangle (){
        super ();
        height = 0;
        width = 0;
    }
    public Rectangle (int theOffset, int theHeight, int theWidth){
        super (theOffset);
        height = theHeight;
        width = theWidth;
    }
    public void set (int newHeight, int newWidth){
        height = newHeight;
        width = newWidth;
    }
    public void drawHere ()   {
        drawHorizontalLine ();
        drawSides ();
        drawHorizontalLine ();
    }
    private void drawHorizontalLine (){
        skipSpaces (getOffset ());
        for (int count = 0 ; count < width ; count++)
            System.out.print ('-');
        System.out.println ();
    }
```

method **drawHere** to draw rectangle

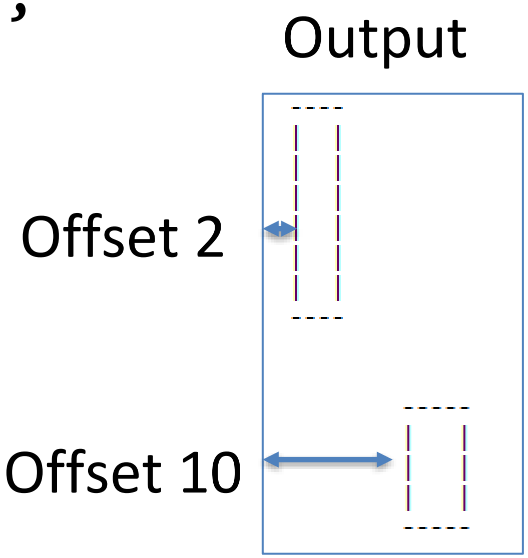
1. Draw the top line
2. Draw the side lines
3. Draw the bottom lines



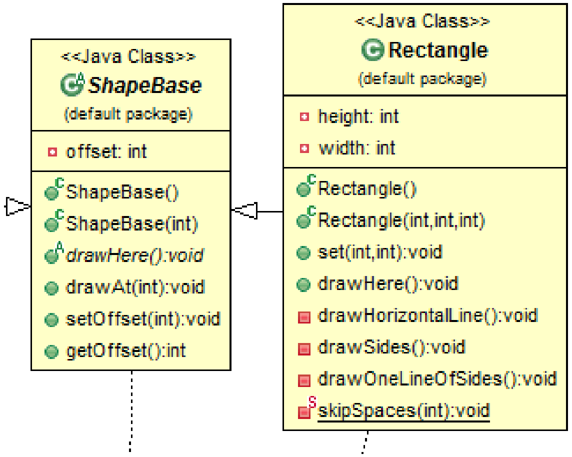
```
private void drawSides (){
    for (int count = 0 ; count <
(height - 2) ; count++)
        drawOneLineOfSides ();
}
private void drawOneLineOfSides (){
    skipSpaces (getOffset ());
    System.out.print ('|');
    skipSpaces (width - 2);
    System.out.println ('|');
}
private static void skipSpaces (int
number){
    for (int count = 0 ; count <
number ; count++)
        System.out.print (' ');
}
}
```

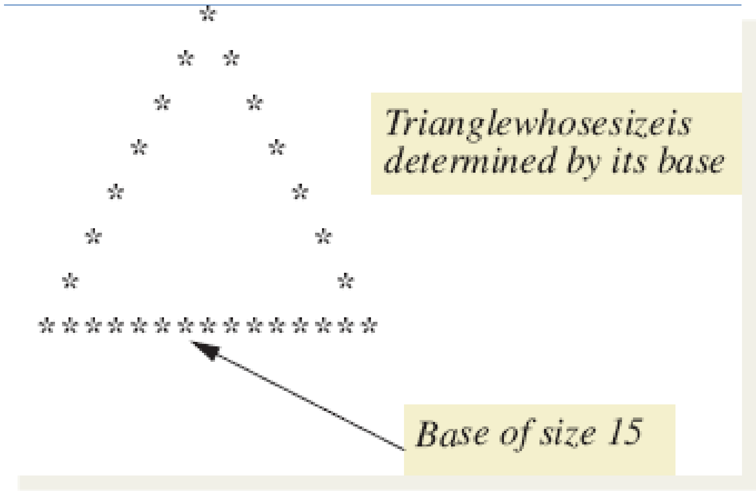
Test 1: draw box

```
public class ShapeDemo{
    public static void main(String[] args) {
    Rectangle box = new Rectangle(2, 8, 4);
     box.drawHere();

     box.set(5, 5);
     box.setOffset(10);
     box.drawAt(2); // 2줄 건너뛰고 그려라
    }
}
```



```java
public class Triangle extends ShapeBasics implements TriangleInterface {
private int base;
public Triangle (){
    super ();
    base = 0; }
public Triangle (int theOffset, int theBase){
    super (theOffset);
    base = theBase;}
public void set (int newBase){     base = newBase; }
public void drawHere () {
    drawTop ();
    drawBase ();  }
private void drawBase () {
    skipSpaces (getOffset ());
    for (int count = 0 ; count < base ; count++)
        System.out.print ('*');
    System.out.println (); }
private void drawTop (){
    int startOfLine = getOffset () + base / 2;
    skipSpaces (startOfLine);
    System.out.println ('*'); //top '*'
    int lineCount = base / 2 - 1; //height above base
    int insideWidth = 1;
    for (int count = 0 ; count < lineCount ; count++)  {
        startOfLine--;
        skipSpaces (startOfLine);
        System.out.print ('*');
        skipSpaces (insideWidth);
        System.out.println ('*');
        insideWidth = insideWidth + 2;
    } }
private static void skipSpaces (int number){
    for (int count = 0 ; count < number ; count++)
        System.out.print (' '); }
}
```





```
       *
      * *
     *   *
    *     *
   *       *
  *         *
 *           *
***************
```

Test 2: draw tree

```
 Save the Redwoods!

               *
              * *
             *   *
            *     *
           *       *
          *         *
         *           *
        *             *
       *               *
      *                 *
     *********************
             ----
             |  |
             |  |
             ----
```

```java
public class TreeDemo {
    public static final int INDENT = 5; // offset
    public static final int TREE_TOP_WIDTH = 21; // odd, base length
    public static final int TREE_BOTTOM_WIDTH = 4; // rectangle w
    public static final int TREE_BOTTOM_HEIGHT = 4; // rectangle h
    public static void main (String [] args)    {
        drawTree (TREE_TOP_WIDTH, TREE_BOTTOM_WIDTH,
                TREE_BOTTOM_HEIGHT);
    }
    public static void drawTree (int topWidth, int bottomWidth, int bottomHeight)    {
        System.out.println (" Save the Redwoods!");
        Triangle treeTop = new Triangle (INDENT, topWidth);
        drawTop (treeTop);
        Rectangle treeTrunk = new Rectangle (INDENT+(topWidth/2)- (bottomWidth / 2),
                                            bottomHeight, bottomWidth);
        drawTrunk (treeTrunk);
    }
    private static void drawTop (TriangleInterface treeTop)    {
        treeTop.drawAt (1);
    }
    private static void drawTrunk (RectangleInterface treeTrunk)    {
        treeTrunk.drawHere (); // or treeTrunk.drawAt(0);
    }
}
```