CISC 3120
C08: Inheritance and Polymorphism

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Recap and issues
  - Project progress? Practice assignments? CodeLab?
  - Review guide #1? Test #1?
  - Automated unit testing?
- Inheritance
- Access control, getters & setters
- Java platform class hierarchy
- Polymorphism via inheritance
- Type casting
- Some discussion on nested classes
- Assignments

Recap: Testing

“Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis. “

-- Alan Perlis

Recap: Unit Testing

- Automated unit tests
  - White-box tests
  - Test coverage (related to case analysis)
  - Separate application logic from tests
  - Automate tests
- JUnit
  - A unit testing framework for Java

Questions?

- Recap and issues
  - Project progress?
  - Practice assignments?
  - CodeLab?
  - Review guide #1?
  - Test #1?
  - Automated unit testing?

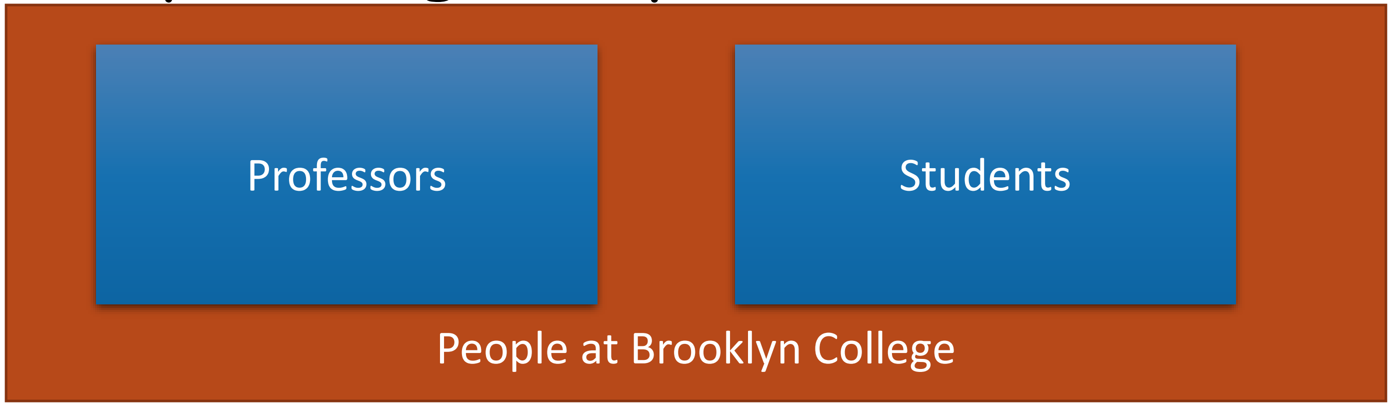
Class and Type

- A class defines a type, and often models a set of entities
- To build a system for managing business at Brooklyn College, we consider
  - People, a set of individuals (objects), modeled as a class that defines the set of objects

People at Brooklyn College

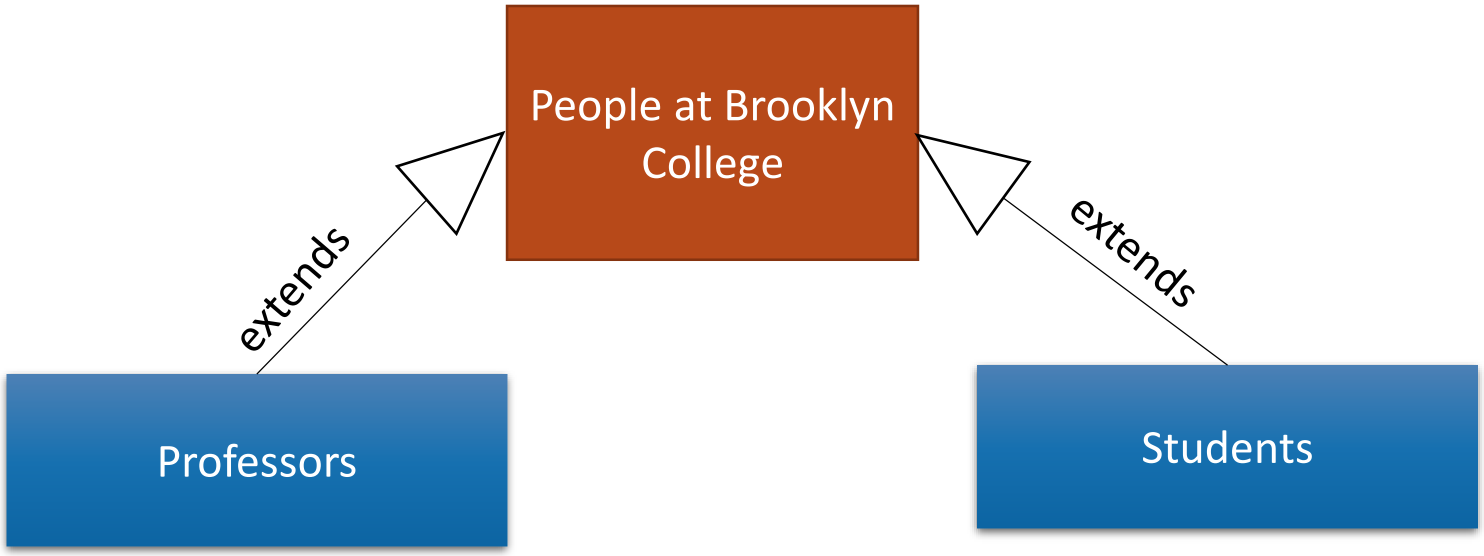
Subtypes

- Some people at Brooklyn are different from the others in some way
- Professors and students are subtypes of Brooklyn College People



Type Hierarchy

- Characteristics and behavior
  - What are Students and Professors in common?
  - What are Students and Professors different?



What's in common?

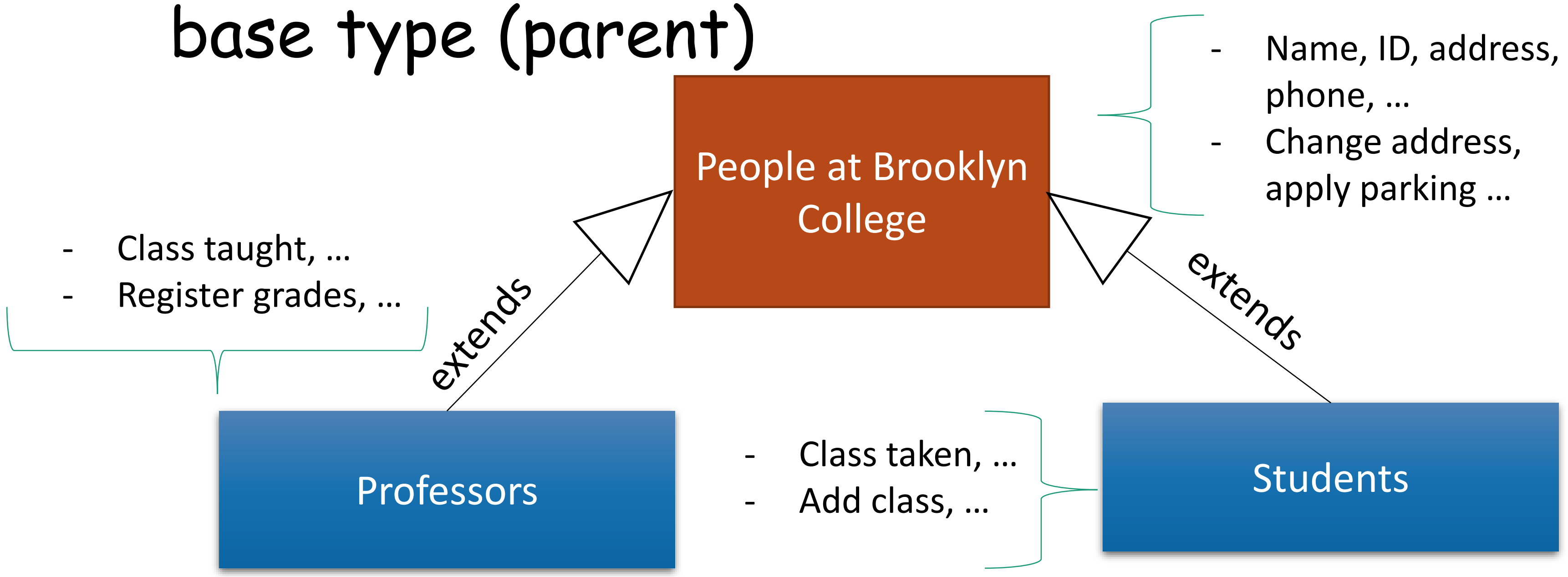
- What characteristics (attributes) and behavior (actions) do People at Brooklyn College have in common?
  - Characteristics (attributes, fields, or states): name, ID, address, email, phone, …
  - Behavior (actions, functions, or methods): change address, apply parking, …

What's Special?

- What's distinct about students?
  - Characteristics (attributes, fields, or states): classes taken, tuition and fees, …
  - Behavior (actions, functions, or methods): add class, drop class, pay tuition, …
- What's distinct about professors?
  - Characteristics (attributes, fields, or states): course taught, rank, title, …
  - Behavior (actions, functions, or methods): register grade, apply promotion, …

Inheritance & Type Hierarchy

- A subtype (child) inherits characteristics (attributes) and behavior (actions) of its base type (parent)



Questions

- Concepts
  - Type, subtype, class, subclass
  - Inheritance

Terms of Choice

- Terms
  - Super type, Super class
  - Base type, Base class
  - Parent type, parent class
  - Child type, child class
  - Subtype, subclass
  - …
- In Java, we sometimes consider “type” and “class” are slightly different
  - In Java, a pure abstract class is called an “interface” (to be discussed in next class)

Questions?

- Terms
  - Super type, Super class
  - Base type, Base class
  - Parent type, parent class
  - Child type, child class
  - Subtype, subclass
  - …

Super Type (Super Class): Person

```
public class Person {
        protected String name;
        protected String id;
        protected String address;
        public Person(String name, String id, String address) {
                this.name = name;  this.id = id;  …
        }
        public void changeAddress(String address) {  …
        }
… }
```

Subtype (Subclass): Student

```
public Student extends Person {
        private ArrayList<String> classesTaken;
        public Student(String name, String id, String address) {
                super(name, id, address);
                classesTaken = new ArrayList<String>();
        }
        public void haveTakenClass(String className) { …
        }
        public void showClassesTaken() { …
        }
…}
```

Subtype (Subclass): Professor

```
public class Professor extends Person {
        private final static int SABATTICAL_LEAVE_INTERVAL = 7;
        private int yearStarted;
        public Professor(String name, String id, String address, int yearStarted) {
                super(name, id, address);
                this.yearStarted = yearStarted;
        }
        public void applySabbatical(int applicationYear) { …
        }
…}
```

Control Access to Members

```
… protected String name; …
```

Modifier	Class	Package	Subclass	World
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
(no modifier)	Yes	Yes	No	No
private	Yes	No	No	No

More restrictive

Choose Access Control Level

- Goal: you want to reduce the chances your class is being misused. Access level is to help achieve it.
  - Use private unless you have a good reason not to.
  - Use the most restrictive access level that makes sense for a particular member.
  - Avoid public fields except for constants. (Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.)

Constructors

- Initialize attributes of an object when it is being created (or instantiated)
- Subclass's constructor
  - Java will call the parent class's **default** constructor if you do not call **one** of parent's constructors explicitly.
    - You may explicitly call it via "super(…)".

```
… super(name, id, address); …
```

Override Methods in Super Class: Methods

```
public class Person { ...
    public String toString() {
       return "Person (name=" + name + ", id=" + id + ", address=" + address + ")";
    } ...
}
```

```
public class Student extends Person { ...
    public String toString() {
         return "Student (name=" + name + ", id=" + id + ", address=" + address
                 + ", coursesTaken=[" + String.join(", ", classesTaken) + "])";
    } ...
}
```

Override Methods in Super Class: Example

```
Person ben = new Person("Ben Franklin", "00124", "2901 Bedford Ave");

Student adam = new Student("Adam Smith", "00248", "2902 Bedford Ave");

System.out.println (ben.toString());

System.out.println(adam.toString());
```

```
Person (name=Ben Franklin, id=00124, address=2901 Bedford Ave)

Student (name=Adam Smith, id=00248, address=2902 Bedford Ave,
coursesTaken=[])
```

Questions

- Inheritance in Java
- Access control of class members
- Constructors
- Overriding methods
- A few other related items
  - this, super

Getters and Setters

- Recall the design principle
  - A class should have only a single responsibility and responsible for its own behavior
  - Objects interacts with only their methods
- How do we access the private members of a class?
  - Getters and setters
    - Getters: a method that returns the value of a restricted variable
    - Setters: a method that sets the value of a restricted variable

Getters and Setters: Example

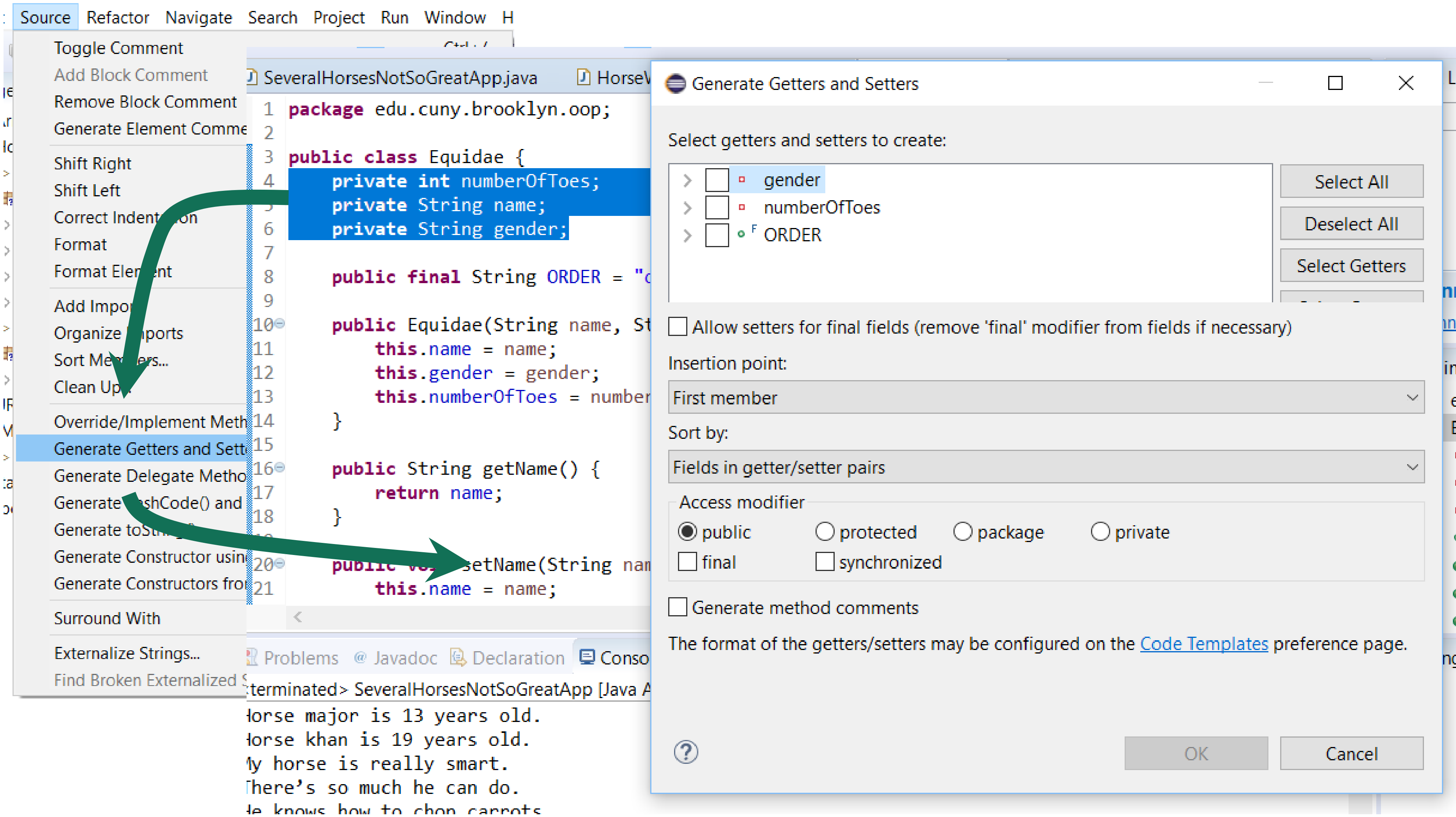
- Observe the getter & setter naming convention

```
public class Boat {
   private String name;
   public String getName() {
       return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

Getters and Setters: Using IDE

- Many IDEs can generate getters and setters for you.
- Examples:
  - In the Eclipse IDE, click the “Source” menu, select “Generate Setters and Getters”

Generating Getters and Setters



```
package edu.cuny.brooklyn.oop;

public class Equidae {
    private int numberOfToes;
    private String name;
    private String gender;

    public final String ORDER = "c

    public Equidae(String name, St
        this.name = name;
        this.gender = gender;
        this.numberOfToes = number
    }

    public String getName() {
        return name;
    }

    public vo   setName(String nam
        this.name = name;
```

<terminated> SeveralHorsesNotSoGreatApp [Java A

```
Horse major is 13 years old.
Horse khan is 19 years old.
My horse is really smart.
There's so much he can do.
He knows how to chop carrots
```

Generate Getters and Setters

Select getters and setters to create:

- ☐ gender
- ☐ numberOfToes
- ☐ ORDER

☐ Allow setters for final fields (remove 'final' modifier from fields if necessary)

Insertion point: First member

Sort by: Fields in getter/setter pairs

Access modifier: [✓ public] protected package private

☐ final ☐ synchronized

☐ Generate method comments

The format of the getters/setters may be configured on the Code Templates preference page.

Questions

- Getters and Setters
- Use IDEs to generate getters and setters

Polymorphism

- One type appears as and is used like another type
- Example
  - A Student object can be used in place of a Person object.
- Inheritance is an approach to realize polymorphism

Polymorphism: Example 1

```
Person ben = new Person("Ben Franklin", "00124", "2901 Bedford Ave");

Person adam = new Student("Adam Smith", "00248", "2902 Bedford Ave");

System.out.println (ben.toString());

System.out.println(adam.toString());
```

```
Person (name=Ben Franklin, id=00124, address=2901 Bedford Ave)

Student (name=Adam Smith, id=00248, address=2902 Bedford Ave,
coursesTaken=[])
```

Polymorphism: Example 2

```
public static void display(Person person) {
	System.out.println(person.toString());
}
```

```
	Person ben = new Person("Ben Franklin", "00124", "2901 Bedford Ave");
	Person adam = new Student("Adam Smith", "00248", "2902 Bedford Ave");
	display(ben); display(adam);
```

```
	Person (name=Ben Franklin, id=00124, address=2901 Bedford Ave)
	Student (name=Adam Smith, id=00248, address=2902 Bedford Ave,
	coursesTaken=[])
```

How about Other Methods?

```
Person ben = new Person("Ben Franklin", "00124", "2901 Bedford Ave");

Student adam = new Student("Adam Smith", "00248", "2902 Bedford Ave");

adam.haveTakenClass("CISC3120");

display(ben); display(adam);
```

```
Person (name=Ben Franklin, id=00124, address=2901 Bedford Ave)

Student (name=Adam Smith, id=00248, address=2902 Bedford Ave,
coursesTaken=[CISC3120])
```

How about this example?

- You say, “adam” appears to be a “Student” object.

```
Person ben = new Person("Ben Franklin", "00124", "2901 Bedford Ave");
Person adam = new Student("Adam Smith", "00248", "2902 Bedford Ave");
adam.haveTakenClass("CISC3120");
display(ben); display(adam);
```

```
Error: The method haveTakenClass(String) is undefined for the type Person
```

Type Casting

- You can only invoke the method of declared type, i.e., Person.

```
Person ben = new Person("Ben Franklin", "00124", "2901 Bedford Ave");

Person adam = new Student("Adam Smith", "00248", "2902 Bedford Ave");

((Student)adam).haveTakenClass("CISC3120");

display(ben); display(adam);
```

```
Person (name=Ben Franklin, id=00124, address=2901 Bedford Ave)

Student (name=Adam Smith, id=00248, address=2902 Bedford Ave,
coursesTaken=[CISC3120])
```

Actual Type and Declared Type

- Declared type: type at compilation time
- Actual type: type at runtime
  - A variable may refer to an object of different type at runtime
  - Example: actual and declared types of “ben”, and “adam”?

```
Person ben = new Person("Ben Franklin", "00124", "2901 Bedford Ave");

Person adam = new Student("Adam Smith", "00248", "2902 Bedford Ave");

((Student)adam).haveTakenClass("CISC3120");
```

Type Casting

- Down-casting
  - Cast to a subtype
  - It is allowed when there is a possibility that it succeeds at run time (e.g., type to be casted to matches actual type)
    - In the example, a “Person” object references to a “Student” object, and the down casting is allowed.
- Up-casting
  - Cast to a super type
  - It is always allowed

Questions

- Polymorphism via inheritance in Java
- Type casting in Java

Design Consideration

- Composition vs. Inheritance

More Example: Boat, RowBoat …

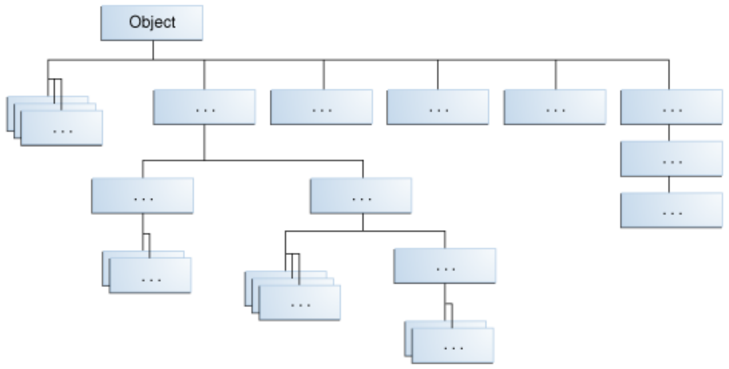
- Both examples (Person-Student-Professor and Boat-RowBoat) are in the “Sample Program” repository on Github

Questions?

- Inheritance or composition?

Java Platform Class Hierarchy

- The java.lang.Object class is the ancestor of all classes
  - defines and implements behavior common to all classes
  - Many classes derive directly from Object
  - Other classes derive from some of those classes, and so on, forming a hierarchy of classe



The Objects class

- java.util.Objects
- Static utility methods for operating on objects.
  - Examples:
    - null-safe or null-tolerant methods for computing the hash code of an object,
    - Methods that return a string for an object
    - Methods that compare two objects.

Questions

- The Java Object and Objects classes

Nested Class

- Java permits one to define a class within another class. Below are 2 of 4 types:
  - Inner class (Non-static nested class)

```
class OuterClass {
    ...
    class NestedClass {   ...   }
}
```

  - Static nested class

```
class OuterClass {
    ...
    static class StaticNestedClass {  ...   }
}
```

Using Nested Class

- Logically grouping classes that are only used in one class
- Can increase encapsulation
- Can lead to more readable and maintainable code

```
class B {
    int c;
}
class A { // B only used in A
    B b = new B();
    b.c = 2;
}
```

```
class A {
    class B {
        int c;
    }
    B b = new B();
    b.c = 2;
}
```

Inner class

- An inner class is a member of the outer class
  - have access to other members of the enclosing class, even if they are declared private.
  - An inner class can be declared private, public, protected, or package private.
  - However, the outer classes can only be declared public or package private

Inner Class: Member of Outer Class

- An instance of the inner class is a part of an instance of the outer class
  - How about create an object of the inner class

Inner Class: Member of Outer Class: Examples

- Which one is correct?

```
class A {
    void method() {
        B b = new B();
    }
   class B { // B only used in A
    }
}
```

```
class A {
   void method() {
        B b = this.new B();
   }
   class B { // B only used in A
   }
}
```

```
class A {
    static void method() {
        B b = new B();
    }
   class B { // B only used in A
    }
}
```

```
class A {
    static void method() {
        A a = new A();
        B b = a.new B();
    }
   class B { // B only used in A  }
}
```

Inner Class: Member of Outer Class: Examples

- Which one is correct?

```
class A {
    void method() {
        B b = new B();
    }
   class B { // B only used in A
    }
}
```

✓

```
class A {
   void method() {
        B b = this.new B();
   }
   class B { // B only used in A
   }
}
```

✓

```
class A {
    static void method() {
        B b = new B();
    }
   class B { // B only used in A
    }
}
```

✗

```
class A {
    static void method() {
        A a = new A();
        B b = a.new B();
   }
   class B { // B only used in A  }
}
```

✓

Static Nested Class

- *A static nested class is associated with its outer class*
    - It belongs to the outer class, not to an object of the outer class.
    - Behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

Static Nest Class: Examples

- Which one is correct or wrong?

```
class A {
    void method() {
        B b = new B();
    }
   static class B { // B only used in A }
}
```

```
class A {
    void method() {
        B b = new A.B();
    }
   static class B { // B only used in A }
}
```

```
class A {
    static void method() {
        B b = new B();
    }
   static class B { // B only used in A }
}
```

```
class A {
    static void method() {
        B b = new A.B();
    }
   static class B { // B only used in A }
}
```

Static Nest Class: Examples

- Which one is correct or wrong?

```
class A {
    void method() {
        B b = new B();
    }
   static class B { // B only used in A }
}
```

✓

```
class A {
    void method() {
        B b = new A.B();
    }
   static class B { // B only used in A }
}
```

✓

```
class A {
    static void method() {
        B b = new B();
    }
   static class B { // B only used in A }
}
```

✓

```
class A {
    static void method() {
        B b = new A.B();
    }
   static class B { // B only used in A }
}
```

✓

Questions?

- Nested classes
  - Inner class
  - Static nested class

Assignments

- Practice Assignment
- CodeLab