香港浸會大學
HONG KONG BAPTIST UNIVERSITY

COMP2026

Problem Solving Using Object Oriented Programming

Inheritance - Part 1

Overview

- Getter, Setter, Default Constructor
- Basic Comparator (Compare Function)
- Basics of ArrayList
- Basics of Inheritance
- Superclass & Subclass
- Constructors in Inheritance
- The `extends` and the `super` keywords
- Access Modifiers in Inheritance
- Overloading & Overriding
- Class Hierarchies

Introduction

- Exam is approaching...
- Let’s develop an examination system
- Displays the questions to students
- Prompts students for their answers
- Checks the expected answer
- Plus... a simple scoring system..
- Oh! We also want a few question types

Question Class (v1)

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Initial Version of the Question Class

```
public class Question {
    private String question;
    private String answer;

    public Question(String question, String answer) {
        this.question = question;
        this.answer = answer;
    }
}
```

- An initial version of the Question class with only question, answer, and a basic constructor

Question Class

Question Class

```
public class Question {
    private String question;
    private String answer;

    public Question(String question, String answer) {
        this.question = question;
        this.answer = answer;
    }
}
```

- Task 1: Add a default constructor to Question
- Task 2: Add getter & setter methods for question & answer
- Task 3: Add a display method for displaying the question
- Task 4: Add a chkAnswer method for checking user's answer

Task 1: Default Constructor

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Default Constructor

```
public Question() {
    this("", "");
}
```

- Default constructor calls the original basic constructor
- More flexible and less likely to introduce bugs
- To call another constructor, do this: `this(...)` with the necessary arguments

Task 2: Helper Methods

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Helper Methods

```
public String getQuestion() {
    return question;
}

public String getAnswer() {
    return answer;
}
```

```
public void setQuestion(String question) {
    this.question = question;
}

public void setAnswer(String answer) {
    this.answer = answer;
}
```

- Add getter & setter methods for `question` & `answer`

Task 3: `display` Method

```
public void display() {
    System.out.println(question);
    System.out.println();
}
```

- `display` displays the question based on the question type

Task 4: the `chkAnswer` Method

The `chkAnswer` method

```
public boolean chkAnswer(String userAnswer) {
    return answer.compareToIgnoreCase(userAnswer) == 0;
}
```

- `chkAnswer` makes use of `compareToIgnoreCase` which compares two String objects (case ignored) and returns an integer:
  - negative integer – the specified string is greater than this string
  - zero – the specified string is equal to this string
  - positive integer – the specified string is less than this string

Task 4: the `chkAnswer` Method

The `chkAnswer` method

```
public boolean chkAnswer(String userAnswer) {
    return answer.compareToIgnoreCase(userAnswer) == 0;
}
```

- As a side note, the `String` class provides:
  - `equals` method and `equalsIgnoreCase` method, which only return a boolean
  - compareTo method and `compareToIgnoreCase` method, which only return an int (as explained on the last slide)
- Typically, classes provide `compareTo` methods if the objects that have some natural order (that is, `objA` is greater than `objB`)

Question Class

- Now, our `Question` class is basically done
- It has basic information:
  - the question
  - the answer
- It has basic functionality:
  - displaying the question
  - checking the answer
- Next: Add the `Exam` class

Exam Class (v1)

Initial Version of the Exam class

```
public class Exam {

}
```

- Task 1: Add `questionList` to Exam (ArrayList)
- Task 2: Add `addQuestion` to `Exam`
- Task 3: Add `prepare` to `Exam`, where we add questions to the `Exam` through `addQuestion`
- Task 4: Add `runIt` to `Exam` for running the exam
- Task 5: Add `promptForAnswer` to `Exam`

Exam Class

Task 1: Adding `questionList`
Add `questionList` to `Exam (ArrayList)`

```
List<Question> questionList = new ArrayList<>();
```

- `ArrayList` works very much like array
- It is a data structure provided by Java for managing a sequence of objects of the same type
- Unlike array, `ArrayList` can grow and shrink as needed
- The more you add to it, the bigger it grows
- After removing elements from it, it shrinks

Task 1: Adding `questionList`

Add questionList to Exam (ArrayList)

```
List<Question> questionList = new ArrayList<>();
```

- ArrayList takes on a funny syntax, as shown above
- `questionList` is an `ArrayList` of `Questions`
- It is declares as `List<Question>`, but initialized as `ArrayList<>()`. This relates to an OO-programming principle, called coding to interfaces (explained later)
- `questionList.add(...)` appends to an ArrayList
- `questionList.get(...)` gets from an ArrayList

Task 2: Adding `addQuestion`

Add `addQuestion` to Exam

```
private void addQuestion(Question question) {
    questionList.add(question);
}
```

- `questionList.add(question)` adds question to the `questionList` - a list of questions
- We will further discuss `ArrayList` in our next chapter

Task 3: prepare and Exam

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Add prepare to Exam

```
public void prepare() {
    Question q1 = new Question("What is the full name of HKBU?",
                               "Hong Kong Baptist University");
    Question q2 = new Question("CS stands for what?",
                               "Computer Science");
    Question q3 = new Question("Are computers smart?",
                               "No!");

    addQuestion(q1);
    addQuestion(q2);
    addQuestion(q3);
}
```

Task 4: Adding `runIt`

- Use a variable to keep track of the score
- Loop through all the questions
- Use `questionList.get(i)` to get a question
- Display the question using `question.display()`
- Prompt for user's answer (using "promptForAnswer")
- Check user's answer using `question.chkAnswer(...)`
- If correct, add one to the score
- After asking all the questions, return the final score to the caller
- In "main", call `runIt()`. Display the score returned by `runIt()`

Task 4: Adding runIt

runIt of Exam

```
public int runIt() {
    int score = 0;

    for (int i = 0; i < questionList.size(); i++) {
        Question question = questionList.get(i);
        question.display();
        String answer = promptForAnswer();
        if (question.chkAnswer(answer)) {
            score++;
        }
    }
    return score;
}
```

- May revise the loop to use enhanced-for loop

Task 4: Adding `runIt`

runIt of Exam

```
Question question = questionList.get(i);
question.display();
String answer = promptForAnswer();
if (question.chkAnswer(answer)) {
    score++;
}
```

- Line 1 gets the question (index i) from questionList
- Note how we use "question.display" to display the question (encapsulation)
- Note how we use "question.chkAnswer" to check the answer (encapsulation)

Task 5: Adding `promptForAnswer`

Task 5: Add `promptForAnswer` to Exam

- Display a prompt (e.g., "Your answer? ")
- Get input from console
- Remove leading and trailing spaces (use trim)
- If input is empty, prompt again
- Otherwise, return the input to caller

Task 5: Adding `promptForAnswer`


```
public String promptForAnswer() {
    Scanner in = new Scanner(System.in);

    String answer = "";
    do {
        System.out.print("Your answer? ");
        answer = in.nextLine();
        answer = answer.trim();
    } while (answer.length() == 0);
    return answer;
}
```

Using `do-while` loop makes much more sense

▶️ Try it!

Exam Class v2

- Now, our Exam class is basically done
- It has basic information:
  - An ArrayList with all of the questions
- It has basic functionality:
  - Adding new questions
  - Running the exam
  - Prompting user for answer
- Next: add a new class `MChoice` (Multiple Choice)

MChoice - Superclass & Subclass

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

- When thinking more deeply, Multiple Choice is actually a type of Questions
- Every MChoice question is a Question

```
public class MChoice extends Question {
    public MChoice(String question) {
        super(question, "");
    }
}
```

`MChoice` - Superclass & Subclass

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

```
public class MChoice extends Question {
    public MChoice(String question) {
        super(question, "");
    }
}
```

- In Object-Oriented programming, this is called inheritance...
  - Question is a **superclass**
  - MChoice is a **subclass**
- Question and MChoice form a **superclass-subclass relationship**
- With **inheritance**, every subclass object is superclass object (e.g., Every `MChoice` question is a `Question`)

MChoice - Superclass & Subclass


```
public class MChoice extends Question {
    public MChoice(String question) {
        super(question, "");
    }
}
```

- In Java, inheritance is represented using "**extends**"...
  - `public class MChoice extends Question`
  - `public class SubClass extends SuperClass`
- The `super` keyword can be used by the subclass to refer to the superclass object
- On line 3, `super(question, "")` would invoke the constructor of the superclass, i.e. it calls:
  `public Question(String question, String answer)`

`MChoice` - Superclass & Subclass

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

```
public class MChoice extends Question {
    public MChoice(String question) {
        super(question, "");
    }
}
```

- Since every subclass object is superclass object, a subclass object has all members of a superclass object
- For example, every `Question` has `question`, `answer`, `display`, `chkAnswer`..., every `MChoice` would have the same
- We say that `MChoice` inherits members of `Question`

`MChoice` - Superclass & Subclass

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

- To access a member from the superclass, you can do something like: super.chkAnswer
- However, for members of superclass, we have access modifiers...
  - `public` members – allow ever body to access
  - `private` members – nobody can access (not even subclass)
  - no modifier (package) – nobody can access except classes from the same package
  - `protected` members – nobody can access except its subclass, and classes from the same package (explain later)

`MChoice` - Superclass & Subclass

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

- Subclass does not have direct access to private members
- Superclass may provide access to its private members through getters or other methods
- Subclass does not directly inherit constructors from superclass (e.g., new MChoice("question", "answer") is not available)
- Subclass can invoke constructors of the superclass via super(...)
- If a subclass constructor does not explicitly call a superclass constructor, the default constructor of the superclass would be called implicitly (error if superclass does not have a default constructor)

`MChoice` - Superclass & Subclass

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

```
public class MChoice extends Question {
    public MChoice(String question) {
        super(question, "");
    }
}
```

A subclass can add more methods to itself. Let's add the following:

- Task 1: `choiceList` (ArrayList for storing choices)
- Task 2: `addChoice(String choice, boolean isTheAnswer)`
- Task 3: `addChoice(String choice)`
- Task 4: `display()`
- Task 5: add a few `MChoice` to `Exam`

Task 1: Adding choiceList

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

The choiceList Class

```
private List<String> choiceList = new ArrayList<>();
```

- We need an `ArrayList`: for storing the choices for the `MChoice`
- Note that the choices (that is, elements of the `ArrayList`) are all `String` objects

Task 2: Adding addChoice (v1)

香港浸會大學 HONG KONG BAPTIST UNIVERSITY

The addChoice Method (v1)

```
public void addChoice(String choice, boolean isTheAnswer) {
    choiceList.add(choice);
    if (isTheAnswer) {
        char theAnswer = (char) ('A' + choiceList.size()-1);
        setAnswer("" + theAnswer);
    }
}
```

- For the choice, we simply add it to `choiceList`
- For the choice is the correct answer, we calculate the choice letter, and store it as the answer using `super.setAnswer`
- As there is no other `setAnswer` in `MChoice`, we can just skip the `super` keyword
- Note how a `char` is used in calculation and converted back

Task 3: Adding addChoice (v2)

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

The addChoice Method (v2)

```
public void addChoice(String choice) {
    addChoice(choice, false);
}
```

- Most choices are not the answer. Let's create a version of `addChoice` method where `isTheAnswer` is default to false
- Object-Oriented Programming is about reusing code (reusability). Let's reuse the previous version of `addChoice`
- On line 2, we just call our previous version of `addChoice` and overload it

Task 4: Overriding display


```
public void addChoice(String choice) {
    addChoice(choice, false);
}
public void display() {
    System.out.println(getQuestion());
    System.out.println();

    for (int i = 0; i < choiceList.size(); i++) {
        char choice = (char) ('A' + i);
        System.out.println(choice + ". " + choiceList.get(i));
    }
    System.out.println();
}
```

- Our superclass, `Question`, has a display method
- Our subclass, `MChoice`, also has a display method, same **signature**
- This is call **method overriding**

Task 4: Overriding display

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Method Overriding

```
Question question  = new Question(...);
MChoice  mchoice = new MChoice(...);

question.display(); // display of Question class is invoked
mchoice.display();  // display of MChoice class is invoked
```

- Our superclass, `Question`, has a `display` method
- Our subclass, `MChoice`, also has a `display` method
- The two methods have the same method signature, which one to invoke?
- Calling display with a `Question` object, `Question.display` would be invoked
- Calling display with a `MChoice` object, MChoice.display would be invoked

Task 5: Add a few `MChoice` to Exam

Modify Exam.prepare to add a few MChoice

```
MChoice mchoice = new MChoice("What is the color of the sky?");
mchoice.addChoice("Red");
mchoice.addChoice("Green");
mchoice.addChoice("Blue", true);
addQuestion(mchoice);
```

- Try adding a few `MChoice` to `Exam` (in the prepare method)
- On line 5, we use `addQuestion(mchoice)`. But `addQuestion` expects `Question`, not `MChoice`!
- `MChoice` is acceptable as every `MChoice` object is a `Question` object!

Try it! ▶️

TrueFalse Class

Initial Version of the TrueFalse class

```
public class TrueFalse extends Question {
    public TrueFalse(String question, boolean ans) {
        super(question, "" + ans);
    }
}
```

- Again, `TrueFalse` a subclass of `Question`
- Note how constructor of TrueFalse handles the answer, `ans` (turning the boolean value into a String)

Work out this on your own!

TrueFalse Class

Initial Version of the `TrueFalse` class

```
public class TrueFalse extends Question {
    public TrueFalse(String question, boolean ans) {
        super(question, "" + ans);
    }
}
```

- Task 1: Add `chkAnswer` to `TrueFalse`
- Task 2: Add a few TrueFalse questions to Exam (via prepare)

TrueFalse Class

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

The chkAnswer Method TrueFalse

```
public boolean chkAnswer(String userAnswer) {
    if (getAnswer().compareTo("true") == 0) {
        return userAnswer.compareToIgnoreCase("t")      == 0 ||
               userAnswer.compareToIgnoreCase("true")   == 0;
    } else {
        return userAnswer.compareToIgnoreCase("f")      == 0 ||
               userAnswer.compareToIgnoreCase("false")  == 0;
    }
}
```

- Note how TrueFalse checks answers received from users
- In the subclass, we can perform very specific actions, tailor made for the subclass itself

Task 2: Add a few TrueFalse to Exam

香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Modify Exam.prepare to add a few TrueFalse

```
TrueFalse tf1 = new TrueFalse("Kevin is kind to everyone.", true);
TrueFalse tf12= new TrueFalse("Assignment 1 is easy for everyone", false);
addQuestion(tf1);
addQuestion(tf2);
```

- Try adding a few TrueFalse to Exam (in the prepare method)

Cars, cArs, cARs, CaRs, cARS...

- In this world, there are many cars
- Suppose we don't just want to talk about cars in general, but something more specific:
  - A mini van to drive your family around
  - A sport car to drive your girlfriend in style
  - A humble little vehicle to drive around town

Similarities and Differences

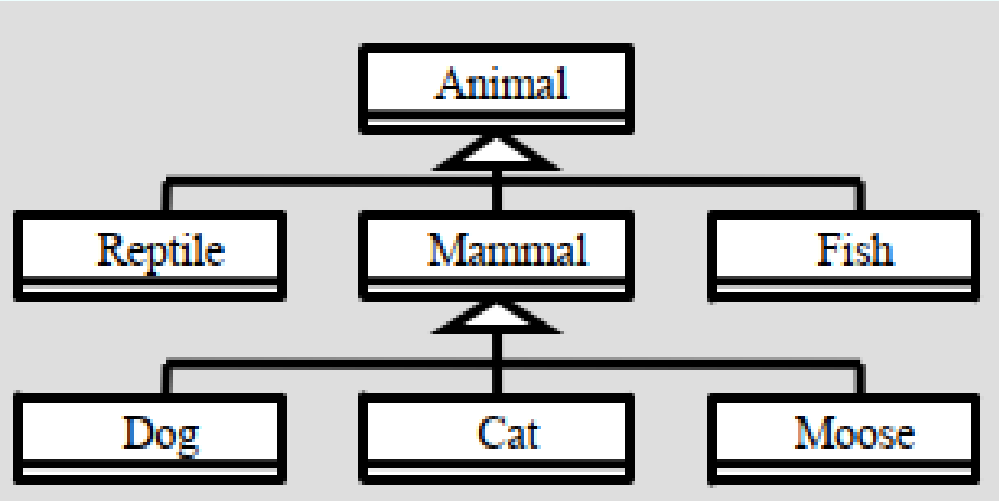
- What do these three automobiles have in common?
  - they’re all vehicles!
  - all can move
  - all have an engine
  - all have doors
  - all have one driver
  - all hold a number of passengers

Similarities and Differences

- What about these three vehicles is different?
  - the sportscar: convertible top, 2 doors, moves really fast, holds small number of people
  - the van: high top, 4 doors (two of which slide open), moves at moderate speed, holds large number of people
  - the CSMobile: normal top, 4 doors, moves slowly, holds moderate number of people

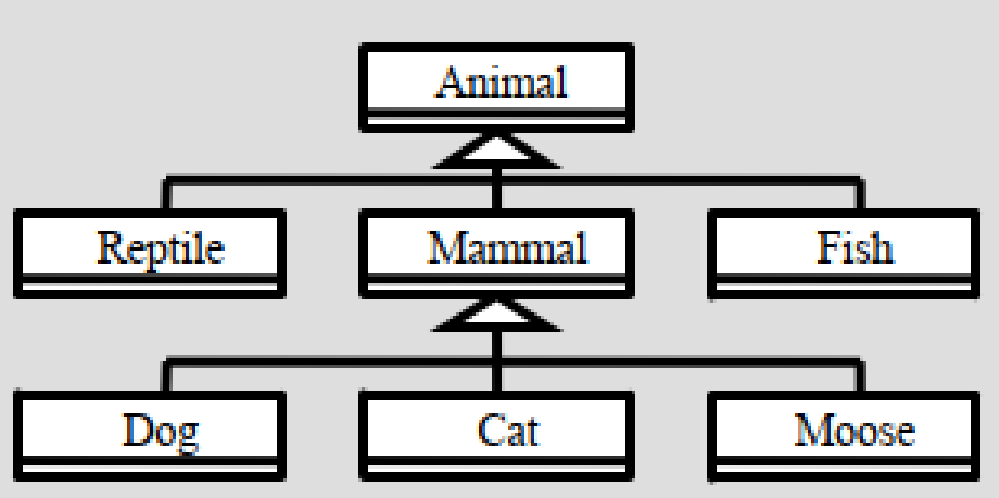
Inheritance

- **Inheritance** models "**is-a**" relationships
  - object “is an” other object if it can behave in the same way
  - inheritance uses similarities and differences to model groups of related objects
- Where there's inheritance, there's an **Inheritance Hierarchy of classes**



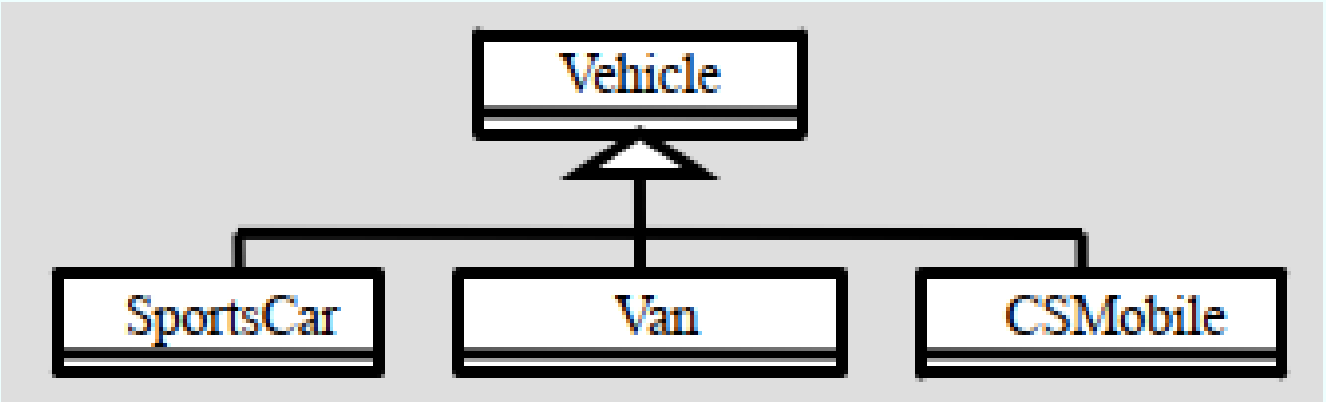
Inheritance

- Inheritance
  - Mammal “is an” Animal
  - Cat “is a” Mammal
- Transitive relationship: a Cat “is an” Animal too
- We can say:
  - Reptile, Mammal and Fish “inherit from” Animal
  - Dog, Cat, and Moose “inherit from” Mammal



Inheritance, Even with Vehicles!

- What does this have to do with vehicles?
  - a SportsCar "**is-a**" Vehicle
  - a CSMobile "**is-a**" Vehicle
  - you get the picture??



- We call this a **tree diagram**, with Vehicle as the "root" and SportsCar, CSMobile, Van as "leaves" (an upside- down tree)
- Let's discuss some important facts about inheritance...

Superclasses and Subclasses

- Inheritance is a way of:
  - organizing information
  - grouping similar classes
  - modeling similarities among classes
  - creating a taxonomy (classification) of objects
- Superclasses – classes higher up in the inheritance hierarchy (e.g., Animal, Mammal)
- Subclass – classes lower in the inheritance hierarchy (e.g., *Reptile*, *Fish*, *Dog*, *Cat*, *Moose*, and even *Mammal*, ...)

Superclasses and Subclasses

- Animal is called superclass
  - a.k.a. base class or parent class
  - in our example, Vehicle is a superclass
- Fish is called subclass
  - a.k.a. derived class or child class
  - in our example, SportsCar is subclass
- A class can be both a superclass & a subclass at the same time
  - e.g., Mammal is superclass of Moose and subclass of Animal
- Can inherit from only one superclass in Java
  - Some programming languages allows a subclass to inherit from multiple superclasses

Inheriting Methods

- Subclass **inherits** all public methods of its superclass
  - if Animals eat and sleep, then Reptiles, Mammals, and Fish eat and sleep
  - if Vehicles move, then SportsCars move!
- Subclass **specializes** its superclass
  - by adding new methods, overriding existing methods, and defining “abstract” methods declared by parent that have no code in them
  - we’ll see these in a few slides!
- Superclass **factors out** methods common among its subclasses
  - subclasses are defined by their differences from their superclass
- Subclass **does not inherit** private methods of its superclass

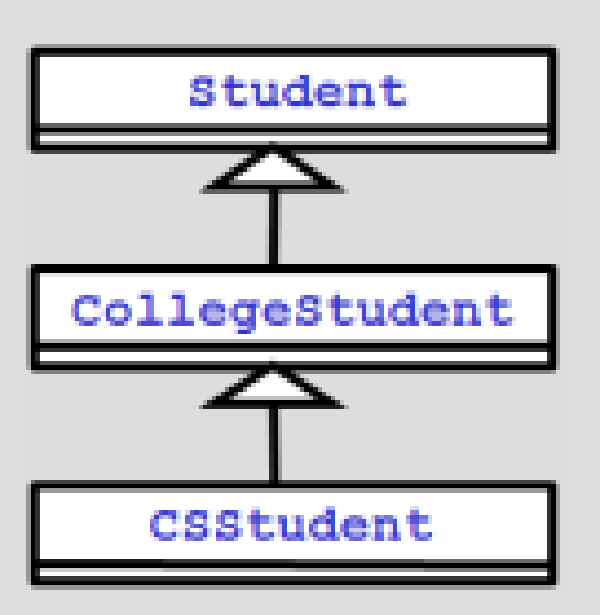
Inheriting Properties

- Subclass **inherits** all public properties (that is, variables) of its superclass, and has direct access to them
- Subclass **inherits** all private properties of its superclass, but has no direct access to them (needs to go through getter/setter)
- Subclass **specializes** its superclass
  - by adding new properties, and overriding existing properties
  - we'll see these in a few slides!
- Superclass **factors out** properties common among its subclasses

Inheriting Capabilities and Properties

- As a general pattern, subclasses:
  - inherit public capabilities (class/instance methods)
  - inherit public properties (class/instance variables)
    - have direct access to them
  - inherit private properties (class/instance variables)
    - do not have direct access to them
    - only indirect access via inherited superclass methods that make use of them
    - for example, accessing them via getter/setter

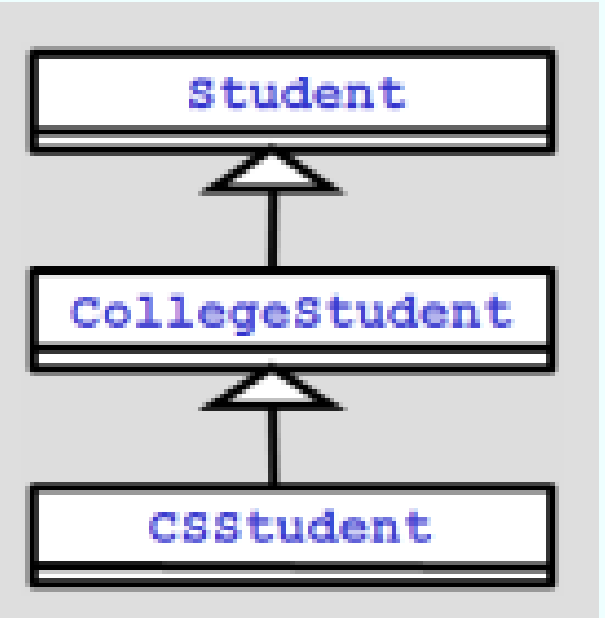
Inheritance Example



- Student inheritance hierarchy:
  - Student is base class
  - CollegeStudent is Student's subclass
  - CSStudent is subclass of CollegeStudent

Inheritance Example

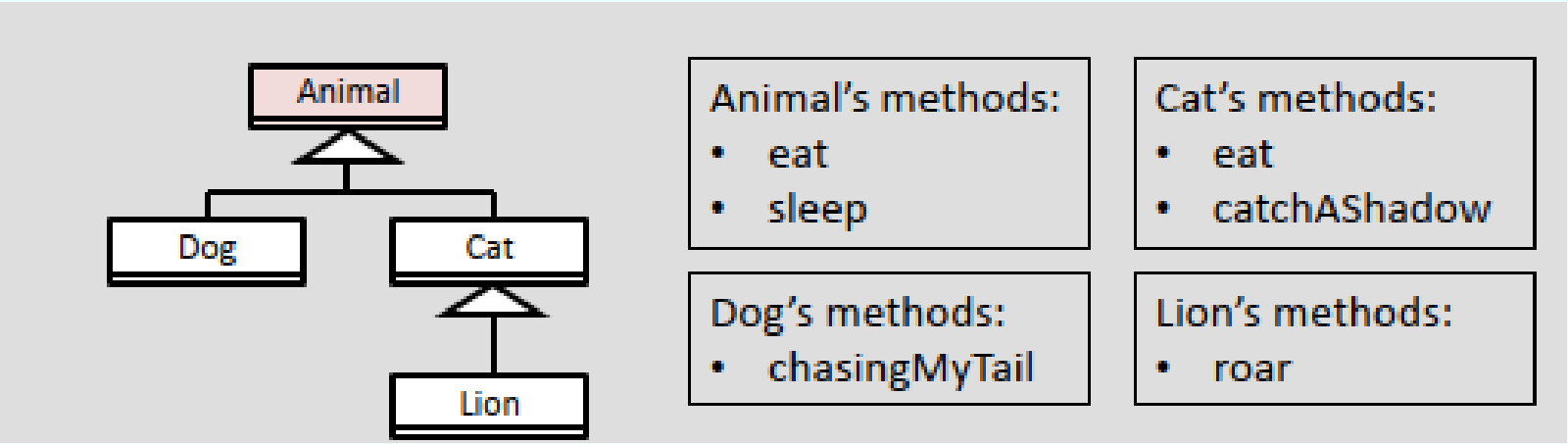
- Student has a capability (or method) `study()` which works by going home, opening a book, and reading 50 pages.
- CollegeStudent "is a" Student, so it inherits the `study()` method, but it overrides the method by:
  - going to the library, reviewing lectures, and doing an assignment
  - note: overriding a method is optional, depending on the design/situation
- Finally, the CSStudent also knows how to `study()` (it `study()` the same way a CollegeStudent does), however, it adds two capabilities
  - `coding()` and `debugging()`



Inheritance Example

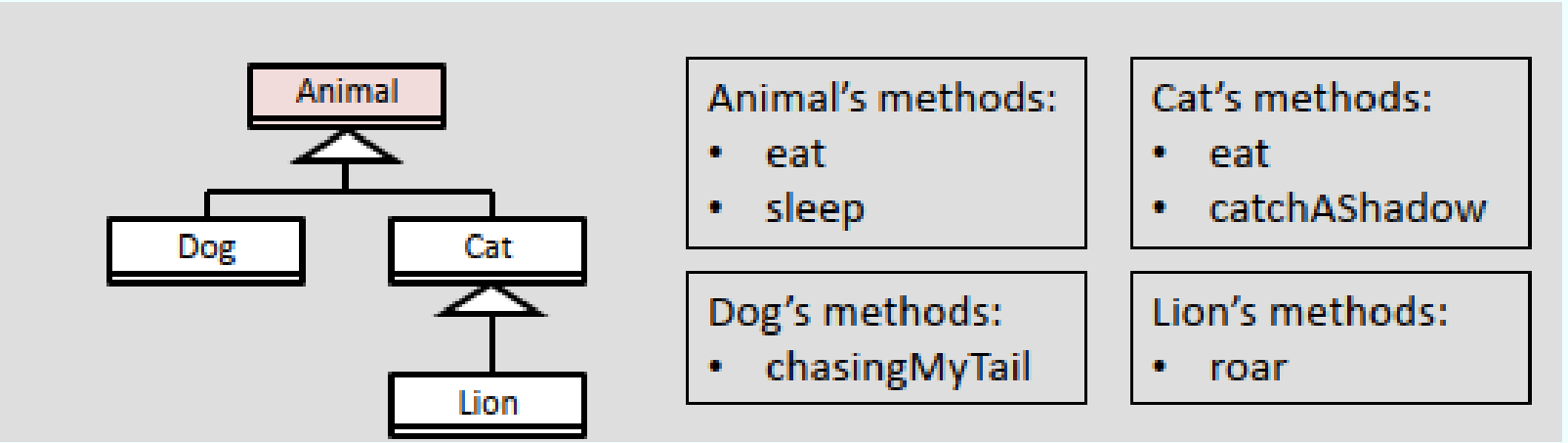
- Each subclass is a **specialization** of its superclass
  - Student knows how to `study()`, so all subclasses in hierarchy know how to `study()`
  - but the CollegeStudent does not `study()` the same way a Student does
  - and the CSStudent has some capabilities that neither Student nor CollegeStudent have (`coding()` and `debugging()`)

Variables Declared as Superclass



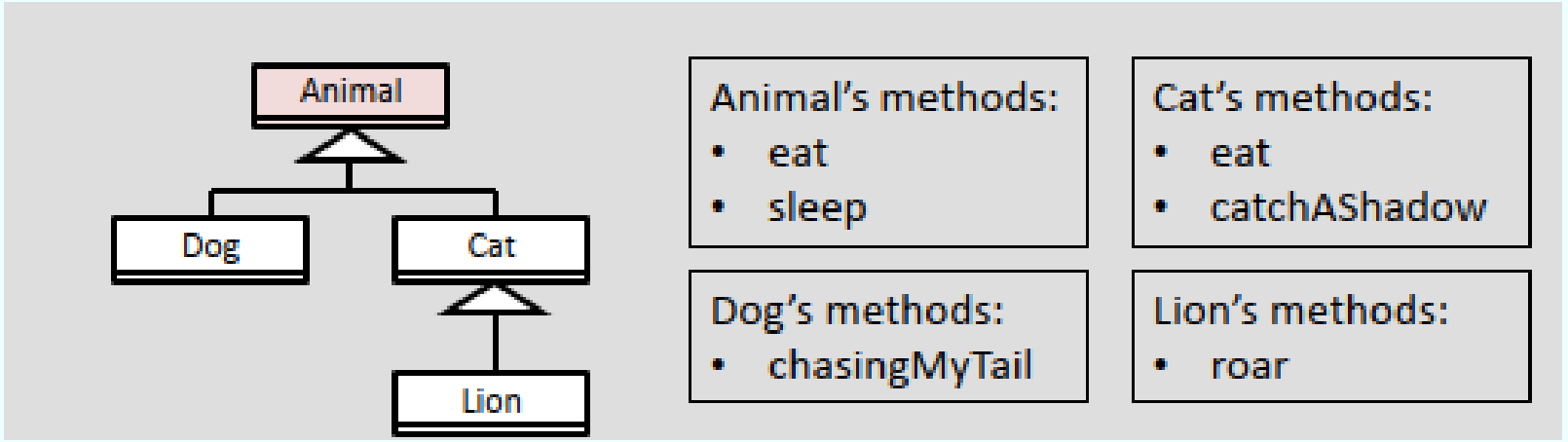
- A variable declared with the type of a superclass can be used for referring to object instances of its subclass
- E.g., a variable declared as Animal can be used for referring to a Dog, a Cat, or a Lion
- E.g., a variable declared as Cat can be used for referring to a Lion

Variables Declared as Superclass



- A variable declared with the type of a superclass can only perform methods available in the superclass
- E.g., a variable declared as Animal, instantiated as...
  - Dog can only eat and sleep like an Animal
  - Cat or Lion can only eat (like a Cat) and sleep (like an Animal)
- E.g., a variable declared as Cat, instantiated as a Lion can only eat (like a Cat), sleep (like an Animal) and catchingShadow (like a Cat)

Variables Declared as Subclass



- A variable declared with the type of a subclass cannot be used for referring to object instances of its superclass
- E.g., a variable declared as Dog or Cat or Lion cannot be used for referring to an Animal
- E.g., a variable declared as Lion cannot be used for referring to a Cat

Variables Declared as Subclass/Superclass

香港浸會大學 HONG KONG BAPTIST UNIVERSITY

```
Animal animal = new Animal();
Cat cat = new Cat();
Dog dog = new Dog();
Lion lion = new Lion();

void method1(Animal animal) {...}
void method2(Cat cat) {...}
void method3(Dog dog) {...}
void method4(Lion lion) {...}
```

Code	Error/No Error
animal = cat	OK
cat = dog;	ERROR!!!
lion = cat	ERROR!!!
cat = lion;	OK
dog = animal;	ERROR!!!

Code	Error/No Error
method1(cat)	OK
method4(dog)	ERROR!!
method1(lion)	OK
method4(cat)	ERROR!!
method4(lion)	OK

Object Construction


```
public class Animal {
  public Animal() {
    System.out.print("Hello Animal...");
  }
}

public class Cat extends Animal {
}

public class Dog extends Animal {
  public Dog() {
    System.out.print("Dog barking...");
  }
}
```

```
Cat cat = new Cat();
Dog dog = new Dog();
Animal animal = new Dog();
Cat cat = new Dog();
```

Code What's the output???