FUNDAMENTALS

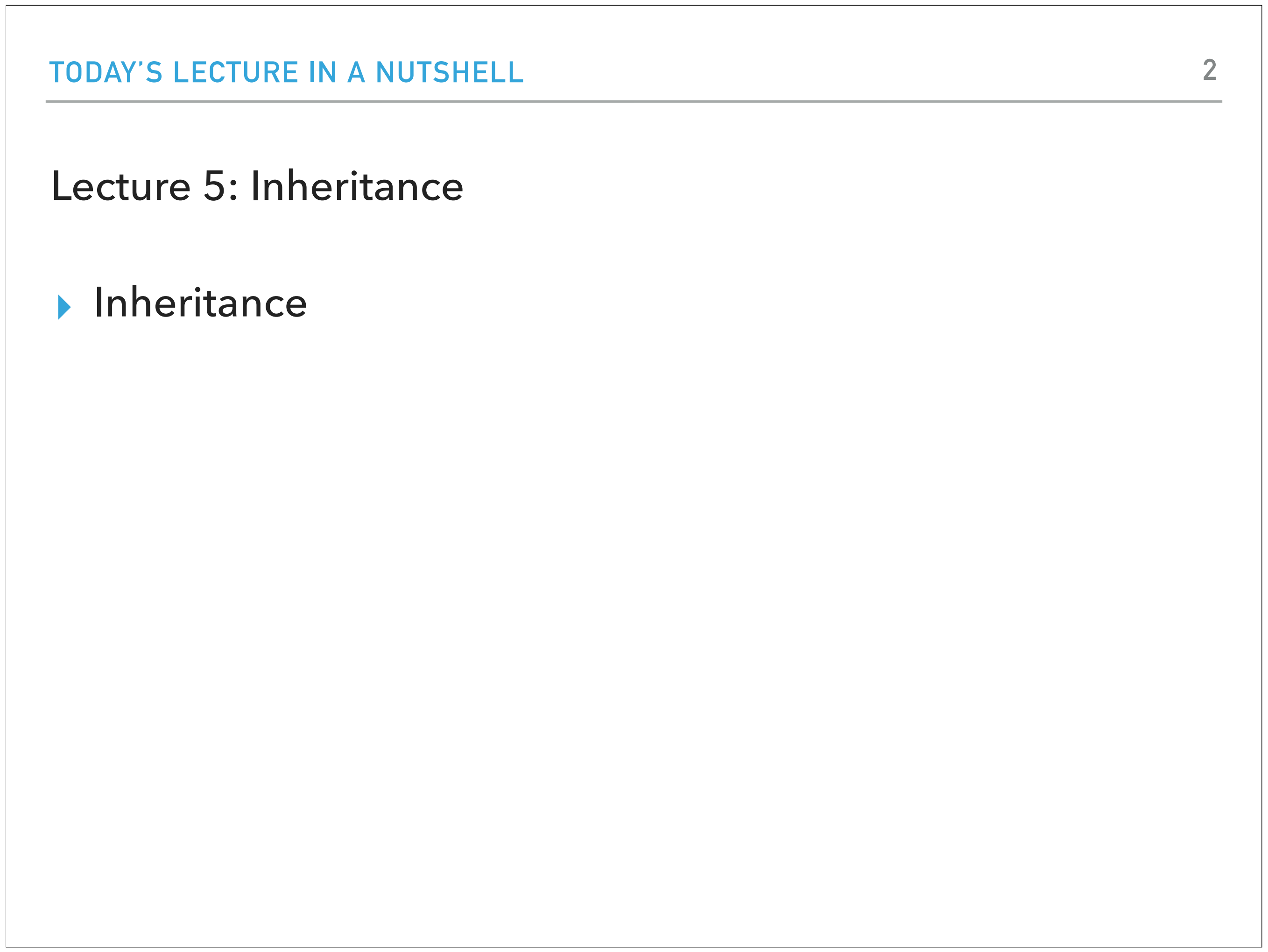
# CS62

## DATA STRUCTURES AND ADVANCED PROGRAMMING

## 5: Inheritance

Alexandra Papoutsaki
she/her/hers

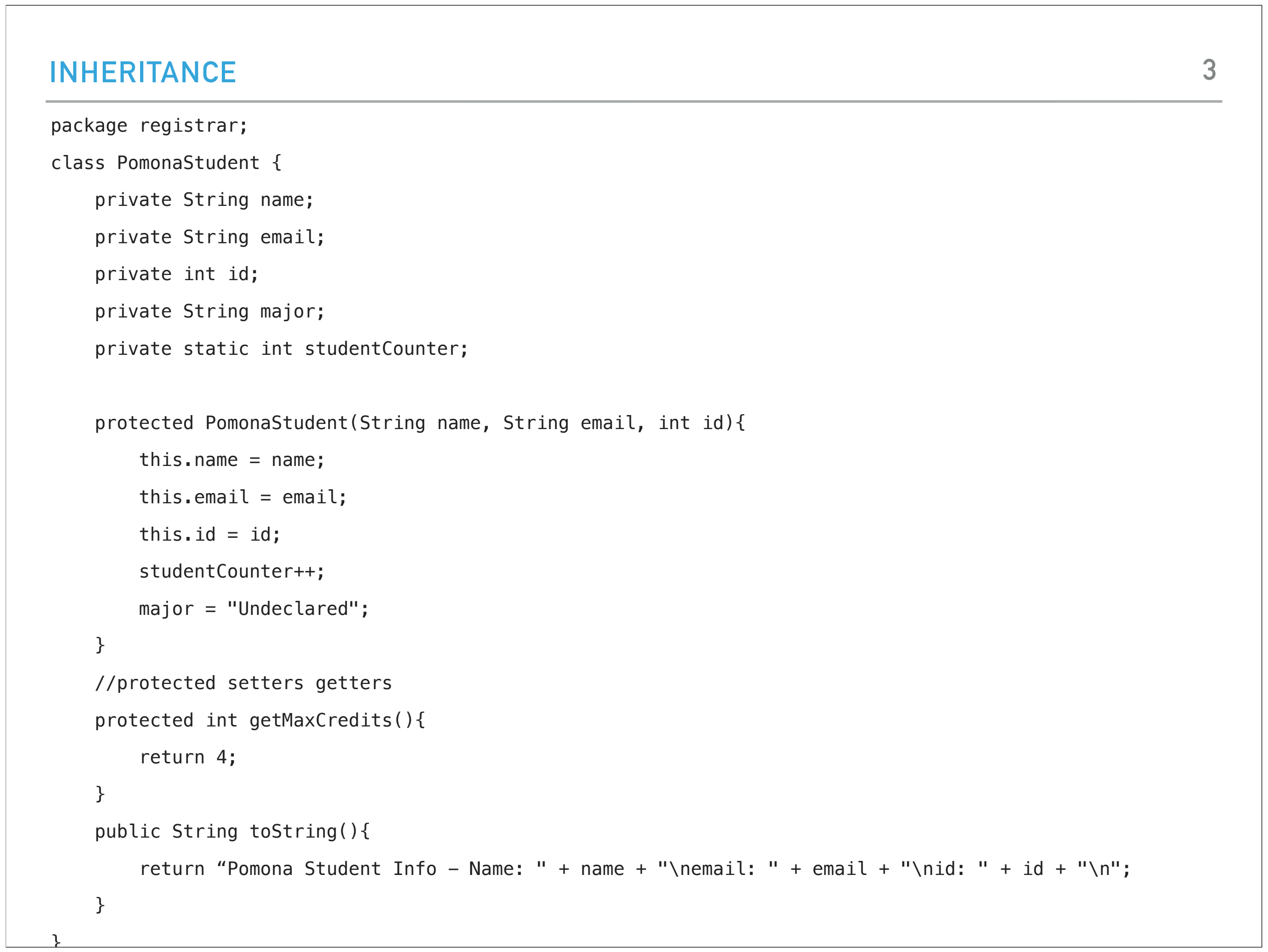
Today we will talk about inheritance, a fundamental concept in object-oriented programming.



## Lecture 5: Inheritance

- Inheritance

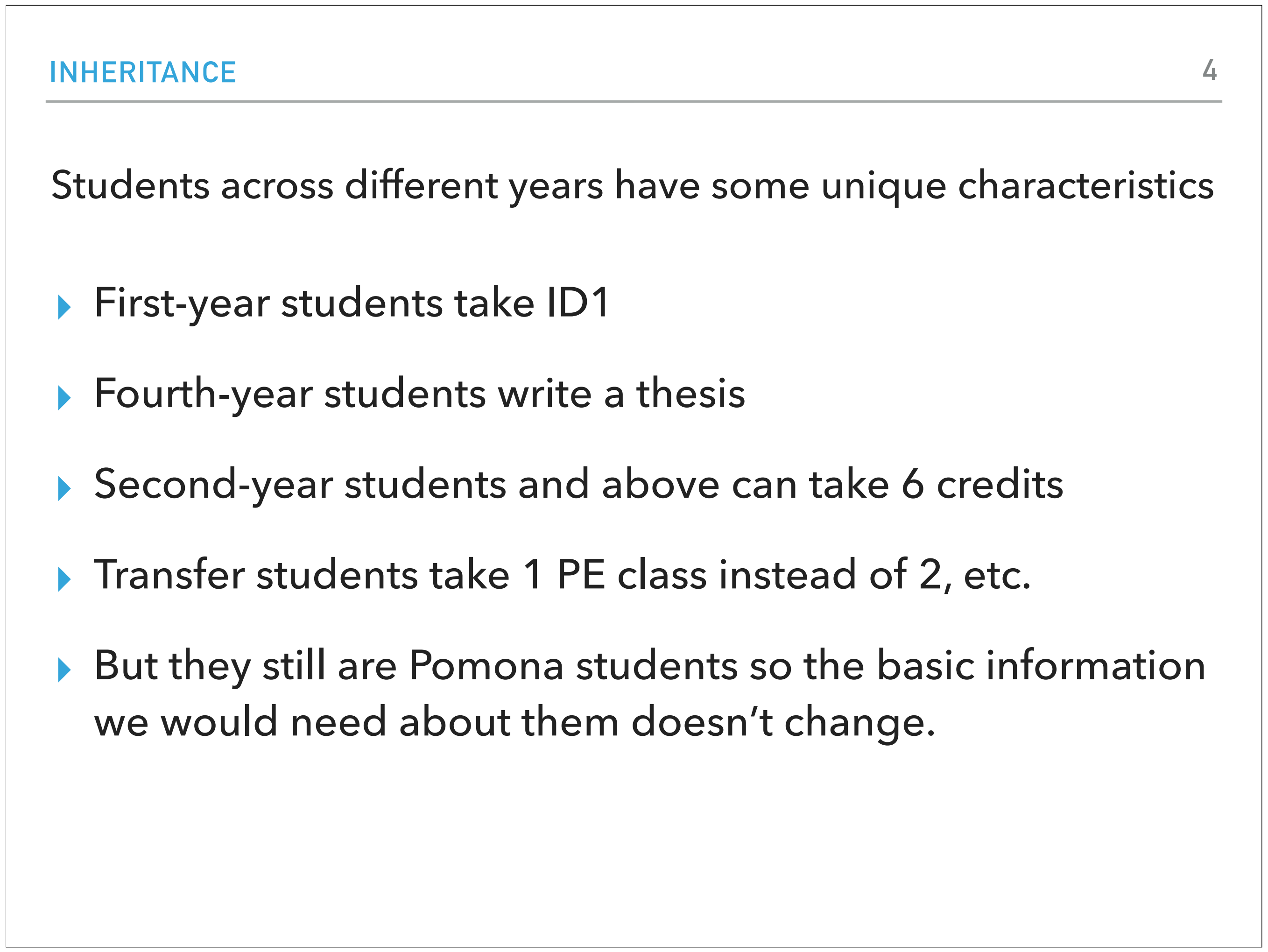
We will use our running example of the PomonaStudent class and your own Dog class to illustrate key concepts.



## INHERITANCE

```
package registrar;
class PomonaStudent {
    private String name;
    private String email;
    private int id;
    private String major;
    private static int studentCounter;

    protected PomonaStudent(String name, String email, int id){
        this.name = name;
        this.email = email;
        this.id = id;
        studentCounter++;
        major = "Undeclared";
    }
    //protected setters getters
    protected int getMaxCredits(){
        return 4;
    }
    public String toString(){
        return “Pomona Student Info - Name: " + name + "\nemail: " + email + "\nid: " + id + "\n";
    }
}
```

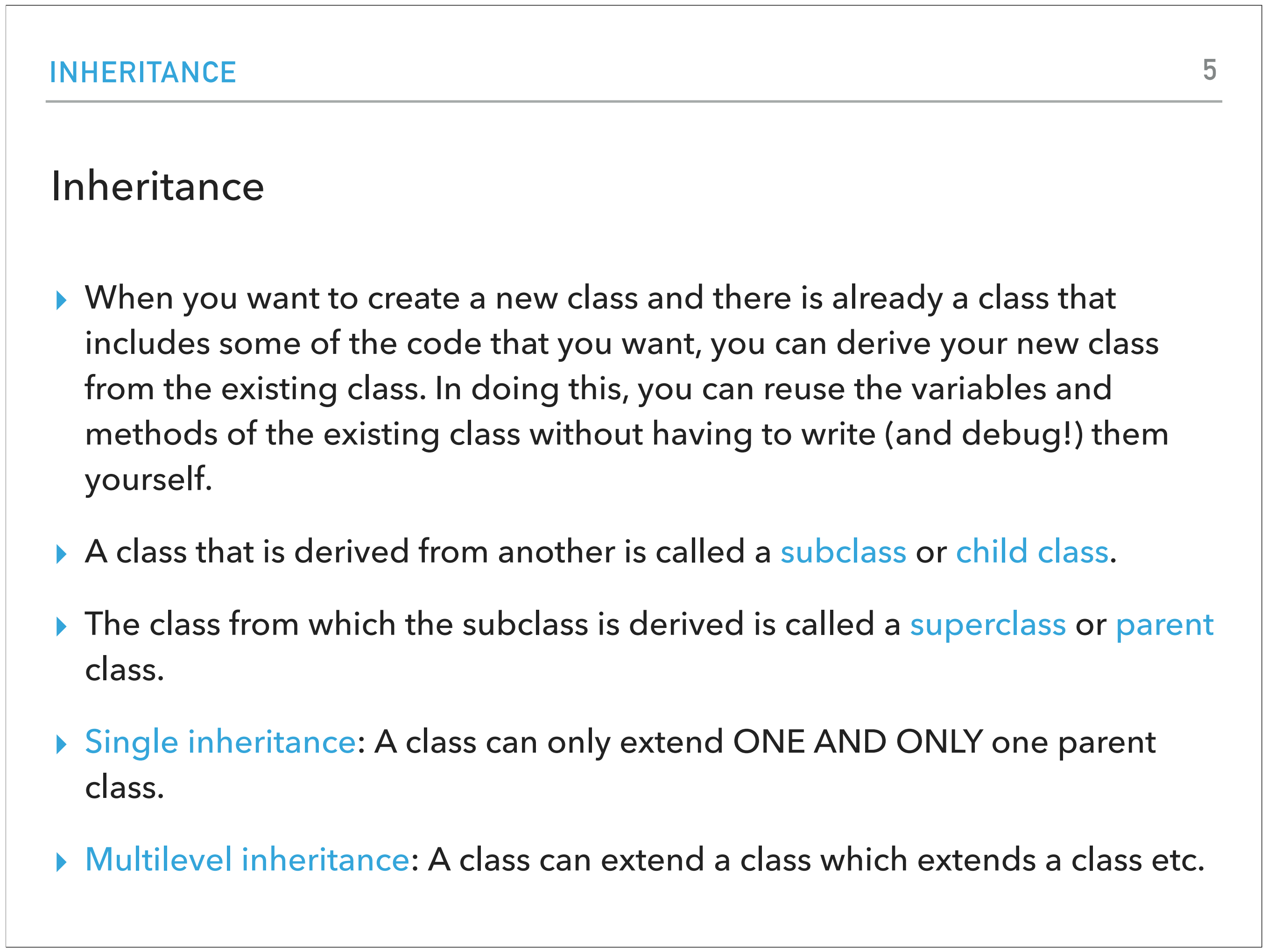
Let’s think of our PomonaStudent class. I have changed it slightly since our last class meeting, mainly by removing some instance variables that won’t be useful in today’s lecture. I have also added one new variable, major, that represents what major the student has declared and which is set by default to “Undeclared” during the instantiation of a PomonaStudent object. I also added an instance method that returns that the default maximum number of credits a student can take is 4.



Students across different years have some unique characteristics

- First-year students take ID1
- Fourth-year students write a thesis
- Second-year students and above can take 6 credits
- Transfer students take 1 PE class instead of 2, etc.
- But they still are Pomona students so the basic information we would need about them doesn't change.

If you think about it, we built a class to represent Pomona students but Pomona students can have unique characteristics based on their year. E.g., first-year students take ID1, fourth-year students write a thesis, second- and above can take 6 instead of 4 credits, transfer students take 1PE class instead of 2 etc (What other unique characteristics can you think of for cohorts of Pomona students? I'd love to enrich the examples shown here). But they still remain Pomona students so the Registrar would like to reuse the code about their id, email, etc. Instead of duplicating our efforts, we can use inheritance.



## Inheritance

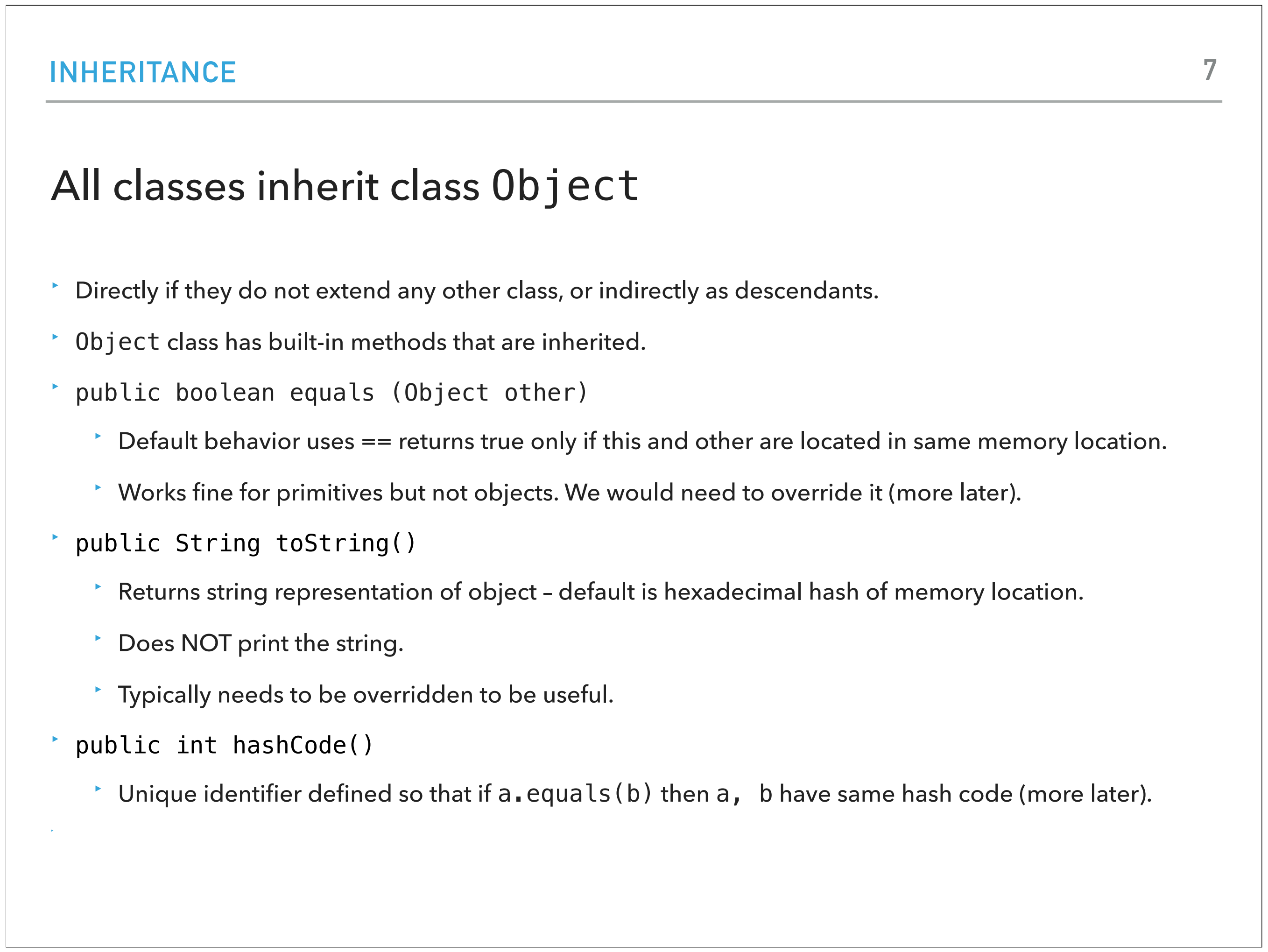
- When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the variables and methods of the existing class without having to write (and debug!) them yourself.
- A class that is derived from another is called a subclass or child class.
- The class from which the subclass is derived is called a superclass or parent class.
- Single inheritance: A class can only extend ONE AND ONLY one parent class.
- Multilevel inheritance: A class can extend a class which extends a class etc.

You can use inheritance when you want to create a new class and there is already a class that includes some of the code that you want. What you do is you derive your new class from the existing class. In doing this, you can reuse the variables and methods of the existing class without having to write (and debug!) them yourself. A class that is derived from another is called a subclass or child class.
The class from which the subclass is derived is called a superclass or parent class.
Single inheritance: In Java, a class can only extend ONE AND ONLY one parent class.
Multilevel inheritance: A class can extend a class which extends a class which itself extends a class etc.

## Inheritance

- The subclass inherits all the `public` and `protected` members.
  - Not the `private` ones, although it can access them with appropriate getters and setters.
- The inherited fields can be used directly, just like any other fields.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.

We talked about access modifiers in Lecture 3. A subclass inherits all public and protected members but not the private ones (it could use though the appropriate getters and setters). The inherited fields can be used directly, just like any other fields. You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.



## All classes inherit class `Object`

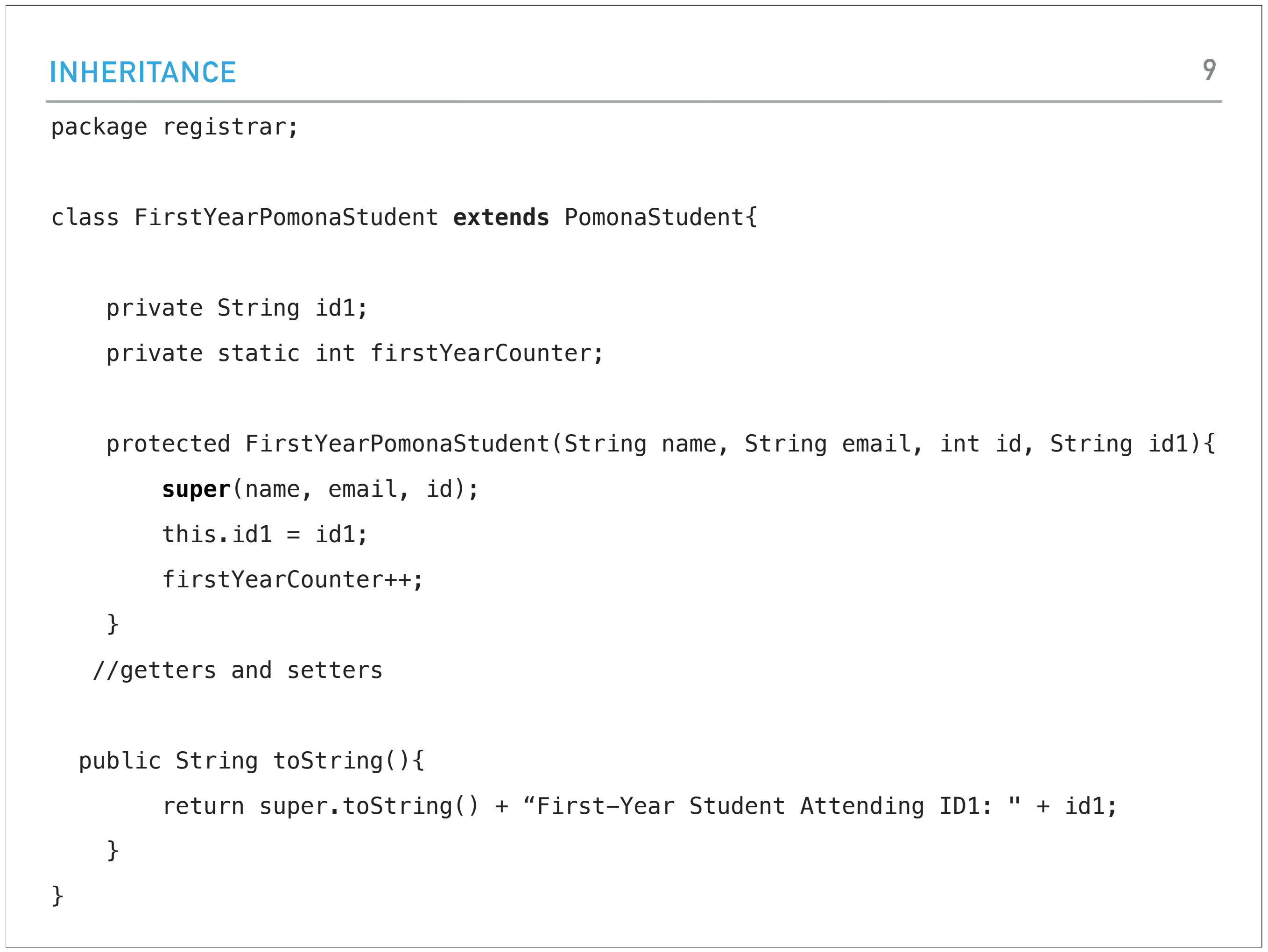
- Directly if they do not extend any other class, or indirectly as descendants.
- `Object` class has built-in methods that are inherited.
- `public boolean equals (Object other)`
  - Default behavior uses == returns true only if this and other are located in same memory location.
  - Works fine for primitives but not objects. We would need to override it (more later).
- `public String toString()`
  - Returns string representation of object – default is hexadecimal hash of memory location.
  - Does NOT print the string.
  - Typically needs to be overridden to be useful.
- `public int hashCode()`
  - Unique identifier defined so that if `a.equals(b)` then `a, b` have same hash code (more later).

All Java classes are direct or indirect descendants of a special class called Object. This applies to both classes written by Java creators and even classes we write ourselves: if we climb up the ladder of inheritance, we will find at some point the class Object. The Object class has three built-in methods that are inherited by all Java classes.
The equals() method compares two objects for equality and returns true only if this and other are located in same memory location. The equals() method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the equals() method.
We have already seen the toString method and we have said that it returns (does NOT print) the string representation of an object (handy when we pass an object to a print statement). We need to override it to be useful. There is one more method, hashCode, which we will revisit at the end of the semester when we talk about dictionaries.

## `super` keyword

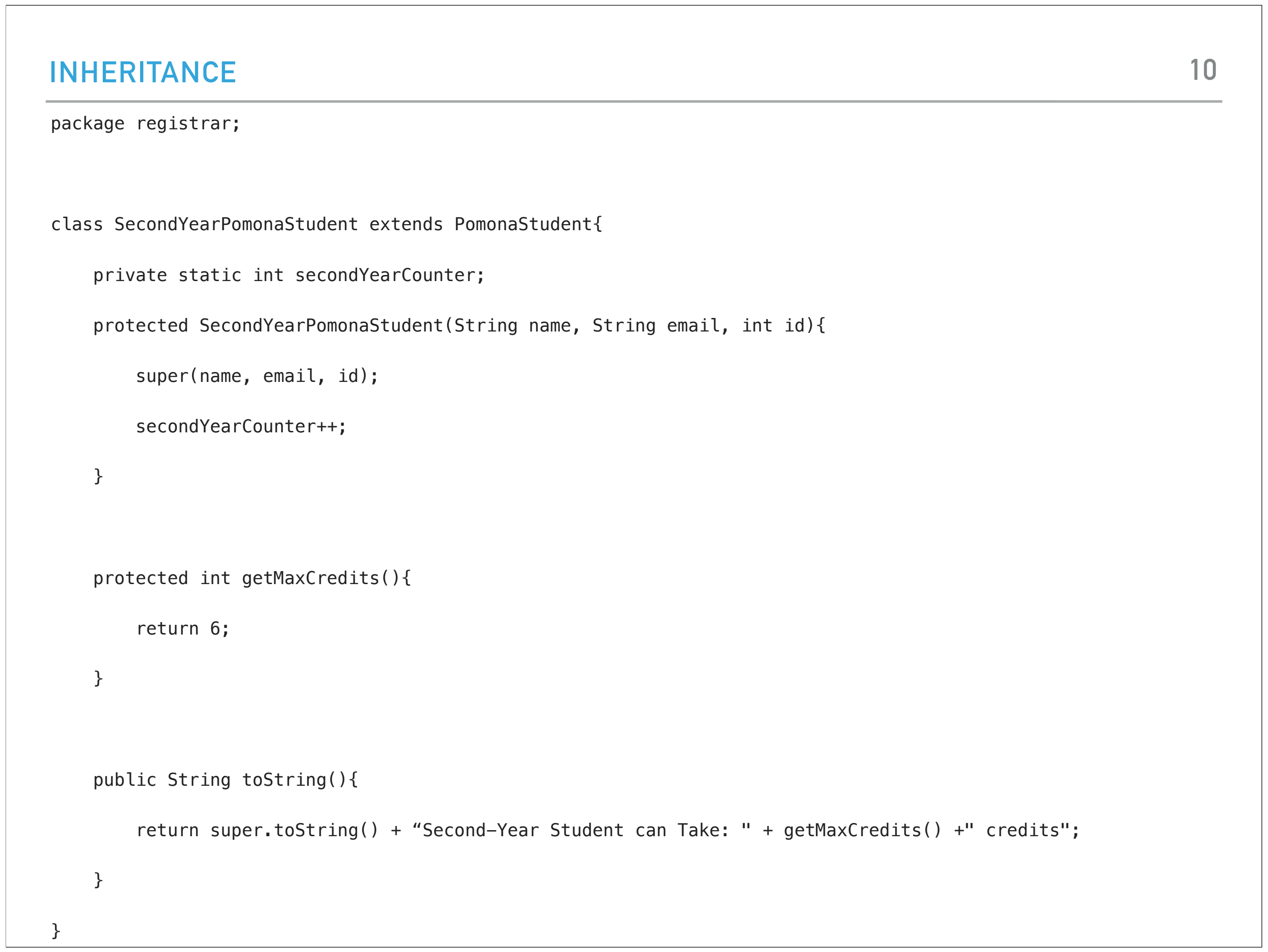
- Refers to the direct parent of the subclass.
- `super.instanceMethod()`: for overridden methods.
- `super(args)`: to call the constructor of the super class. First line in constructor of subclass.

If you remember, we said that the this keyword refers to the current object. Similarly, the super keyword refers to the direct parent of a class. We can use super to invoke a parent's overriden method. And we can also use super(args) to call the constructor of the super class. In fact, this is done by default for us by Java which invokes the no-argument constructor of the parent, but we will get in the habit of always having as the first line of a subclass constructor a call to its parent's constructor.



```
package registrar;

class FirstYearPomonaStudent extends PomonaStudent{

    private String id1;
    private static int firstYearCounter;

    protected FirstYearPomonaStudent(String name, String email, int id, String id1){
        super(name, email, id);
        this.id1 = id1;
        firstYearCounter++;
    }
   //getters and setters

  public String toString(){
        return super.toString() + “First-Year Student Attending ID1: " + id1;
    }
}
```

Let’s see how that plays out in the class FirstYearPomonaStudent which extends PomonaStudent. Here you will see that I have added one new field that corresponds to the ID1 class a first year takes and a static counter for first year students. The constructor of FirstYearPomonaStudent, calls the constructor of PomonaStudent and updates the ID1 class. The toString method, calls the toString of the PomonaStudent class and expands this message to add some information about the ID1.



## INHERITANCE

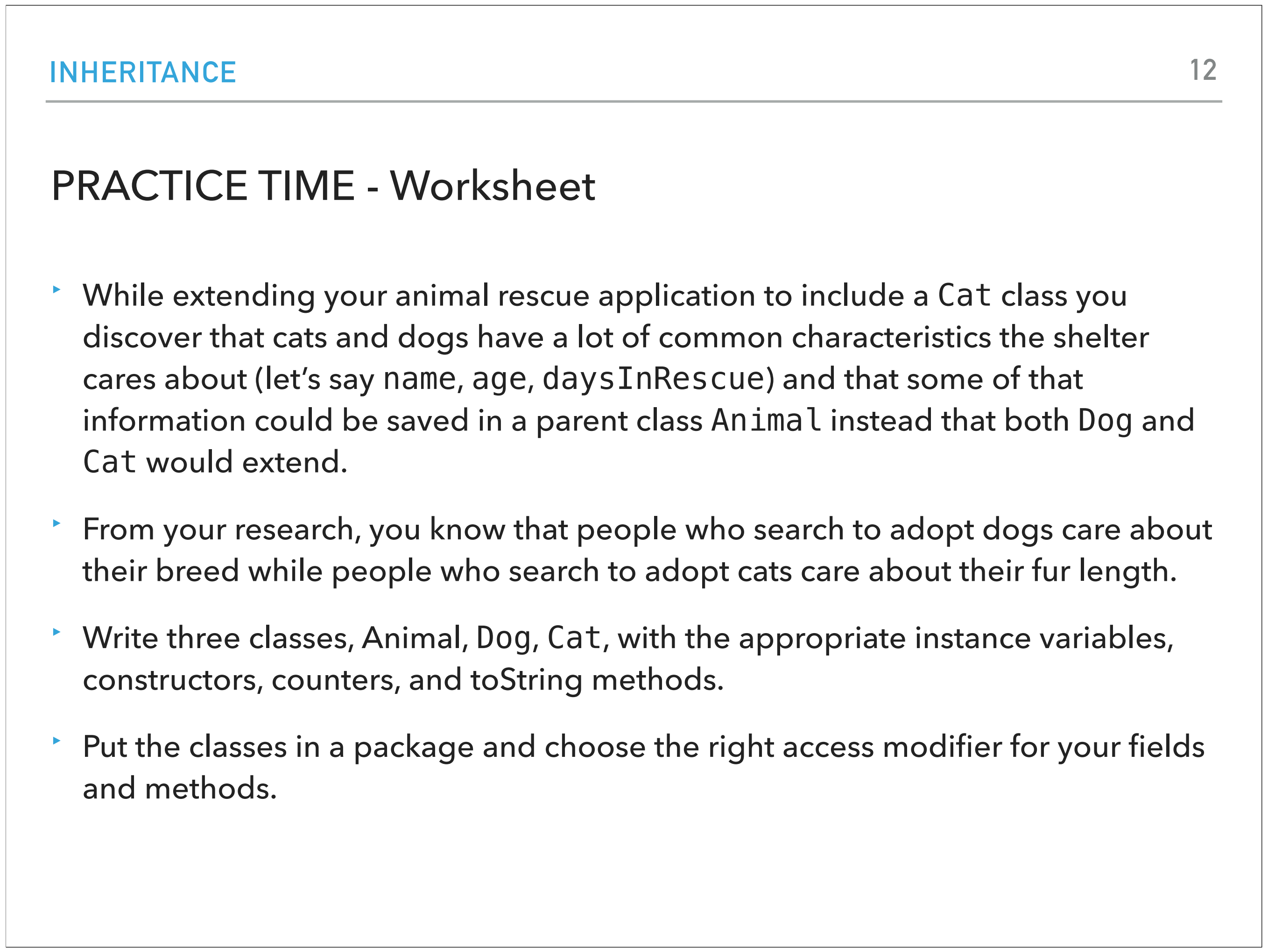
```
package registrar;


class SecondYearPomonaStudent extends PomonaStudent{

    private static int secondYearCounter;

    protected SecondYearPomonaStudent(String name, String email, int id){

        super(name, email, id);

        secondYearCounter++;

    }


    protected int getMaxCredits(){

        return 6;

    }


    public String toString(){

        return super.toString() + “Second-Year Student can Take: " + getMaxCredits() +" credits";

    }

}
```

Similarly for a second year student, I updated its toString. I couldn’t come up with any unique field other than a counter (I am open to suggestions) but I thought that the getMaxCredits should be changed to return 6 now.

## INHERITANCE

```
package registrar;


class FourthYearPomonaStudent extends PomonaStudent{

    private String thesisTitle;

    private static int fourthYearCounter;

    protected FourthYearPomonaStudent(String name, String email, int id, String thesisTitle){

        super(name, email, id);

        this.thesisTitle = thesisTitle;

    }

    //getters and setters

   protected int getMaxCredits(){

        return 6;

    }

    public String toString(){

        return super.toString() + "Fourth-Year Student Writing Thesis on: " + thesisTitle;

    }

}
```

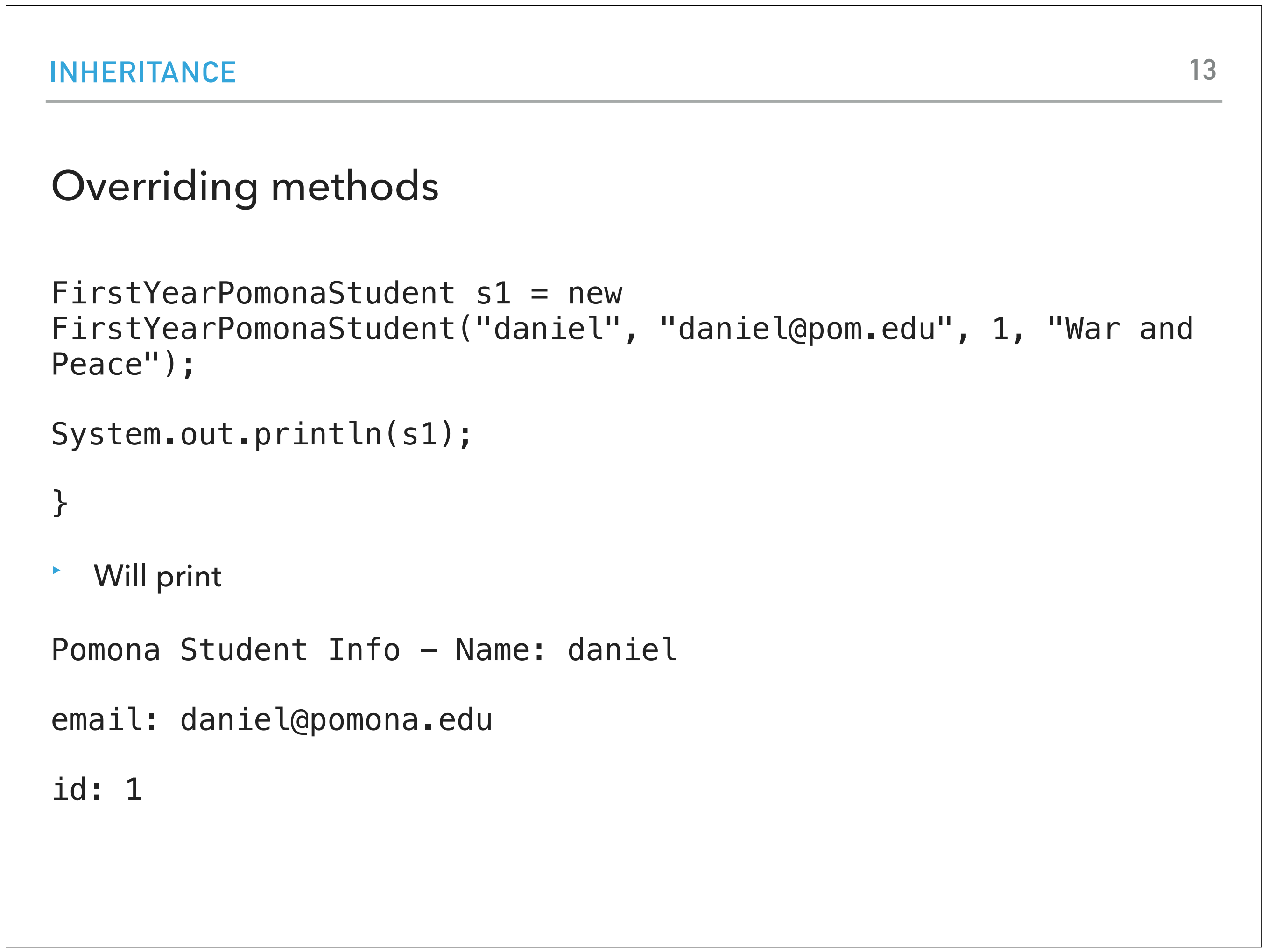
As a last example, here is the FourthYearPomonaStudent class which has an additional thesis title field and counter, max credits at 6, and an updated toString.



## PRACTICE TIME - Worksheet

- While extending your animal rescue application to include a `Cat` class you discover that cats and dogs have a lot of common characteristics the shelter cares about (let's say `name`, `age`, `daysInRescue`) and that some of that information could be saved in a parent class `Animal` instead that both `Dog` and `Cat` would extend.
- From your research, you know that people who search to adopt dogs care about their breed while people who search to adopt cats care about their fur length.
- Write three classes, Animal, `Dog`, `Cat`, with the appropriate instance variables, constructors, counters, and toString methods.
- Put the classes in a package and choose the right access modifier for your fields and methods.

While extending your animal rescue application to include a Cat class you discover that cats and dogs have a lot of common characteristics the shelter cares about (let's say name, age, daysInRescue) and that some of that information could be saved in a parent class Animal instead that both Dog and Cat would extend.
From your research, you know that people who search to adopt dogs care about their breed while people who search to adopt cats care about their fur length.
Write three classes, Animal, Dog, Cat, with the appropriate instance variables, constructors, counters, and toString methods.
Put the classes in a package and choose the right access modifier for your fields and methods.



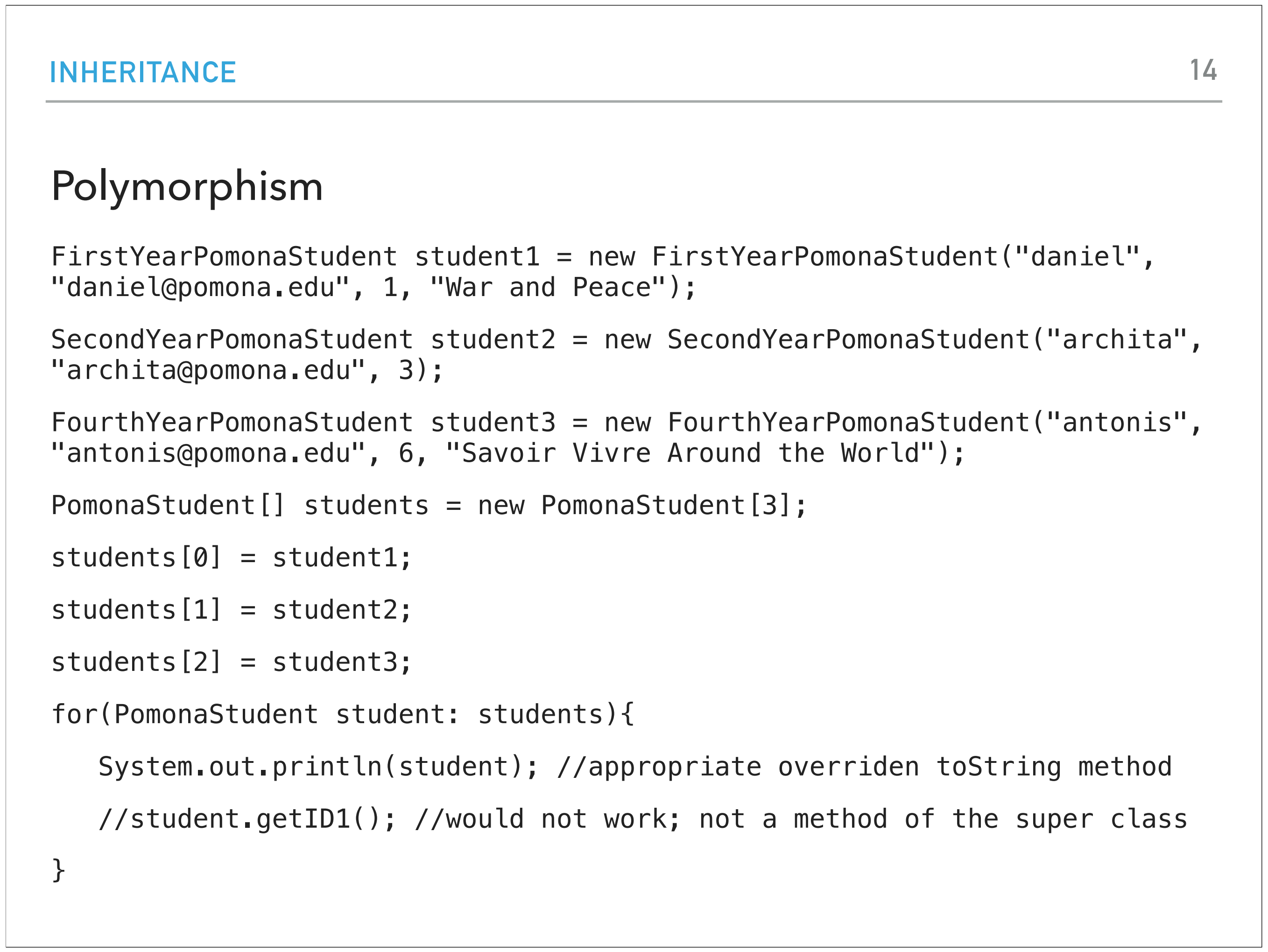
## Overriding methods

```
FirstYearPomonaStudent s1 = new
FirstYearPomonaStudent("daniel", "daniel@pom.edu", 1, "War and
Peace");

System.out.println(s1);

}
```

- Will print

```
Pomona Student Info - Name: daniel

email: daniel@pomona.edu

id: 1
```

Let’s assume we have a FirstYearPomonaStudent object and we print it: we will see a combination of the subclass and superclass tostring message since we called the super.toString. We can take that a step further with polymorphism.



## Polymorphism

```
FirstYearPomonaStudent student1 = new FirstYearPomonaStudent("daniel",
"daniel@pomona.edu", 1, "War and Peace");

SecondYearPomonaStudent student2 = new SecondYearPomonaStudent("archita",
"archita@pomona.edu", 3);

FourthYearPomonaStudent student3 = new FourthYearPomonaStudent("antonis",
"antonis@pomona.edu", 6, "Savoir Vivre Around the World");

PomonaStudent[] students = new PomonaStudent[3];

students[0] = student1;

students[1] = student2;

students[2] = student3;

for(PomonaStudent student: students){

   System.out.println(student); //appropriate overriden toString method

   //student.getID1(); //would not work; not a method of the super class

}
```

Polymorphism, another central concept in OOP, allows an object to take different forms. Let's say, I have here three objects, one of each of the subclasses of PomonaStudent. I can make an array of PomonaStudents, and Java would allow me to add them because not only they are FirstYearPomonaStudents, they also are PomonaStudents. If I try to print each student object, the appropriate overridden toString method would be invoked. The cool thing is that this ensures that I cannot access methods that are specific to only one subclass, e.g., the getID1.

## Polymorphism

- `ParentClass obj = new ChildClass();` E.g.,

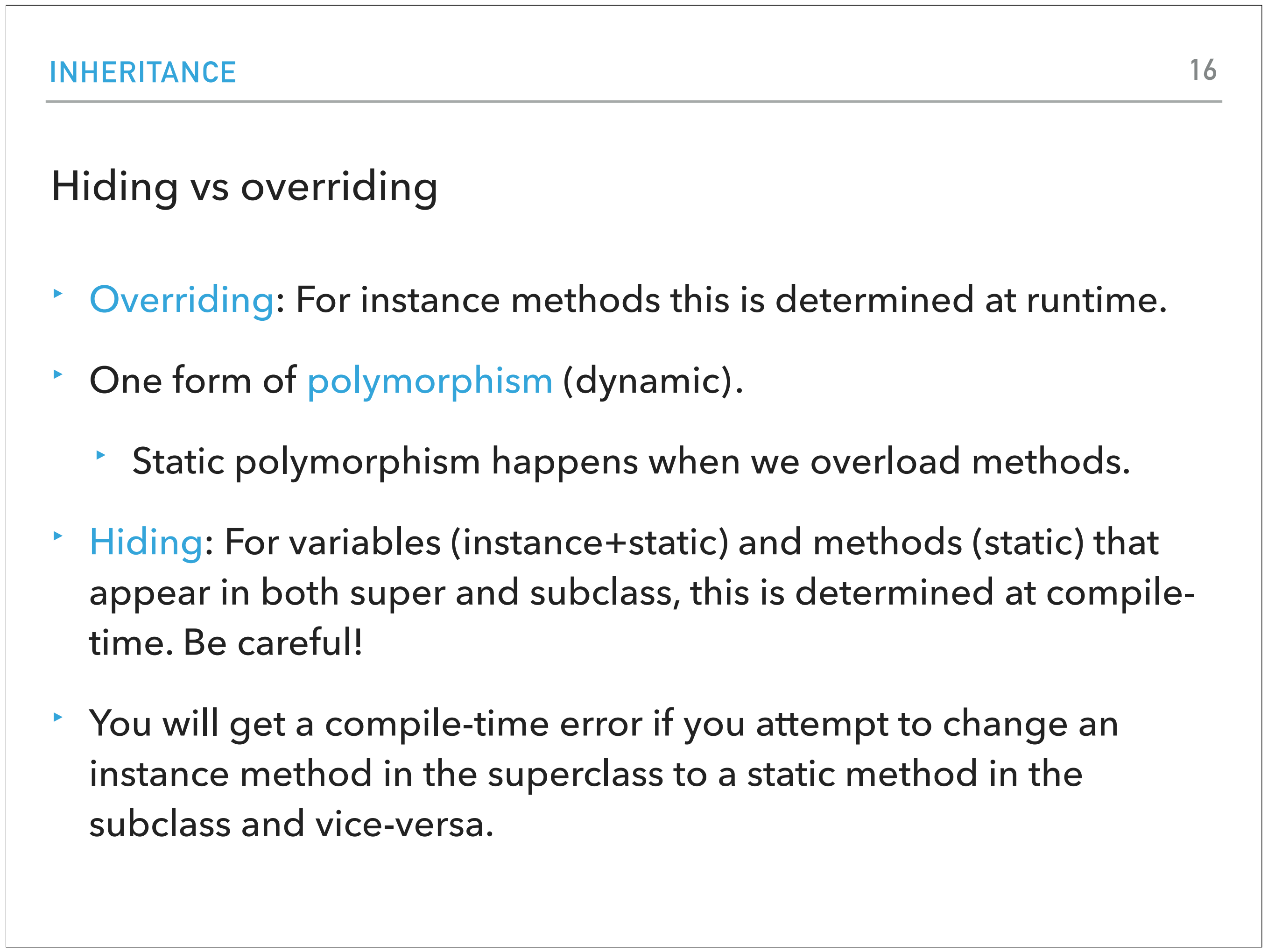
```
PomonaStudent student7 = new FirstYearPomonaStudent("alex",
"alex@pomona.edu", 1, "Humans through the eyes of technology”);

System.out.println(student7.getMaxCredits());
```

- Will print 4

```
 student7 = new SecondYearPomonaStudent(student7.getName(),
student7.getEmail(), student7.getId());
```

- Will print 6

Java allows a bit of an unusual syntax, ParentClass obj = new ChildClass().  In this example, we have child class object assigned to the parent class reference. In order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference). If you assign a parent type to a subclass it means that you agree with to use the common features of the parent class. It gives you the freedom to abstract from different subclass implementations. As a result limits you with the parent features.



## Hiding vs overriding

- Overriding: For instance methods this is determined at runtime.
- One form of polymorphism (dynamic).
  - Static polymorphism happens when we overload methods.
- Hiding: For variables (instance+static) and methods (static) that appear in both super and subclass, this is determined at compile-time. Be careful!
- You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass and vice-versa.

In contrast to overriding which represents one of the forms of polymorphism (dynamic which is determined at runtime; the other one, static, happens when we overload methods), we also have hiding. This happens when variables (either instance or static) and static methods have the same name in the super and subclass. The class would be determined at compile time so you need to be very careful. Note that you will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass and vice-versa.

## Example: `Animal`

```
public class Animal {
    public int legs = 2;
    public static String species = "Animal";
    public static void testStaticMethod() {
        System.out.println("The static method in Animal");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Animal");
    }
}
```

Let’s say I have this class Animal with two variables (one instance: legs, one static: species) and one static and one instance method.

## Example: Cat

```
public class Cat extends Animal {
    public int legs = 4;
    public static String species = "Cat";
    public static void testStaticMethod() {
        System.out.println("The static method in Cat");
    }
    public void testInstanceMethod() {
        System.out.println("The instance method in Cat");
    }
}
```

Now let’s assume that the class Cat extends Animal and has exactly the same names for the variables and methods, with different contents.

## Hiding vs overriding

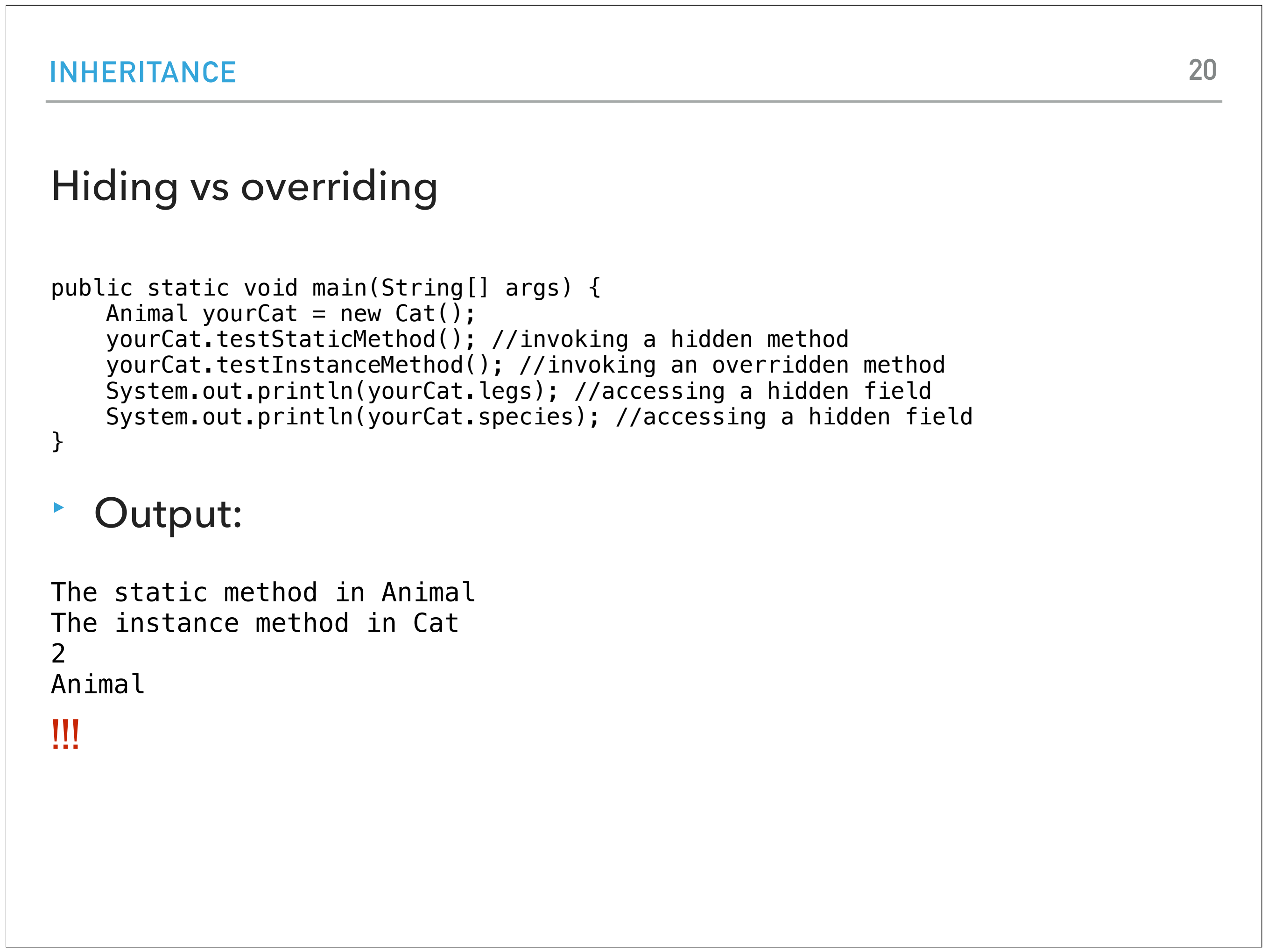
```
public static void main(String[] args) {
    Cat myCat = new Cat();
    myCat.testStaticMethod(); //invoking a hidden method
    myCat.testInstanceMethod(); //invoking an overridden method
    System.out.println(myCat.legs); //accessing a hidden field
    System.out.println(myCat.species); //accessing a hidden field
}
```

- Output:

```
The static method in Cat
The instance method in Cat
4
Cat
```

**WHAT YOU WERE EXPECTING, RIGHT?**

So far, if we make a Cat object and call the different fields and methods the output is as we expect.



## Hiding vs overriding

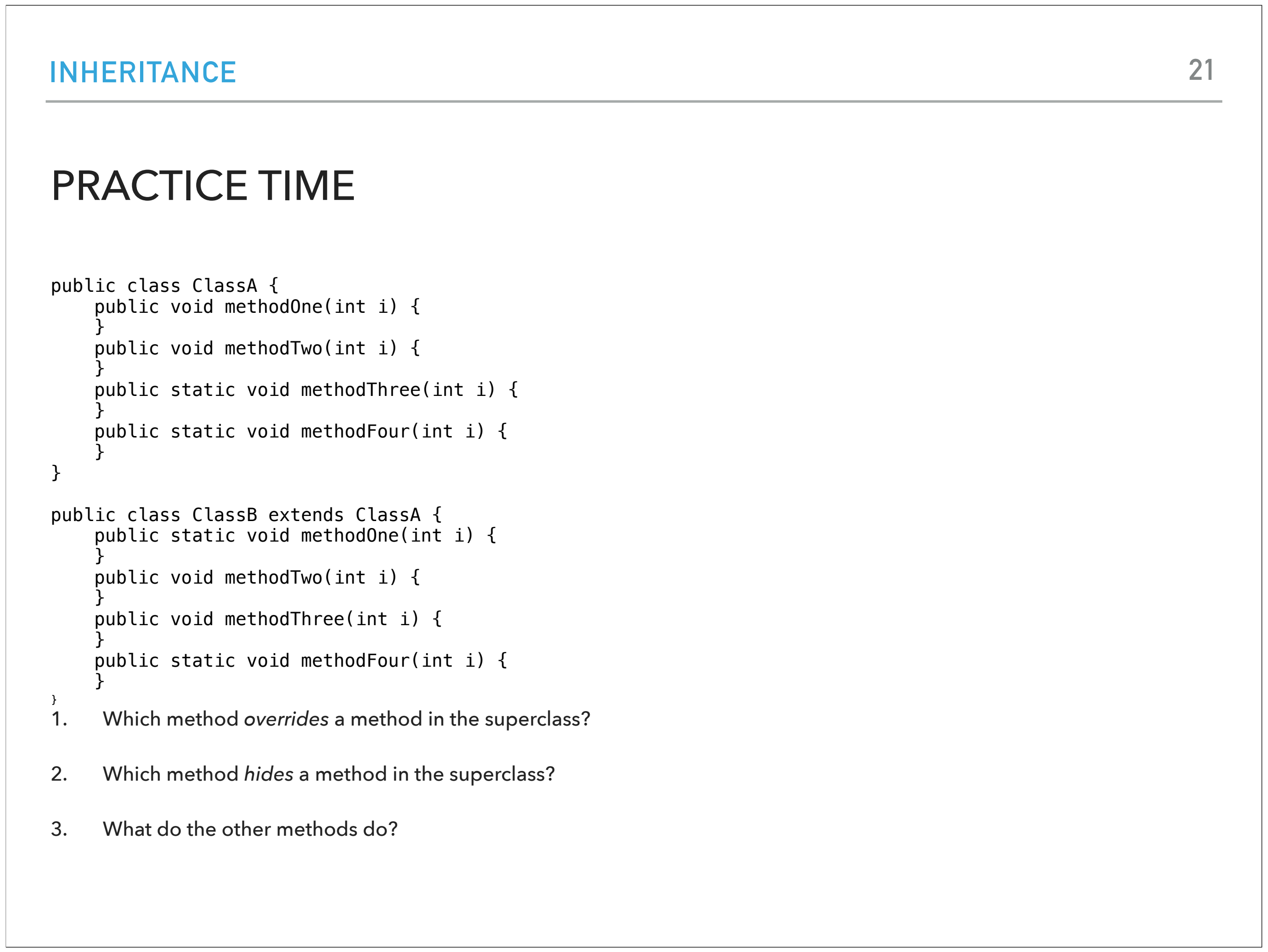
```
public static void main(String[] args) {
    Animal yourCat = new Cat();
    yourCat.testStaticMethod(); //invoking a hidden method
    yourCat.testInstanceMethod(); //invoking an overridden method
    System.out.println(yourCat.legs); //accessing a hidden field
    System.out.println(yourCat.species); //accessing a hidden field
}
```

- Output:

```
The static method in Animal
The instance method in Cat
2
Animal
```

!!!

In contrast, if we assign the parent type Animal to the subclass Cat and then call yourCat.testStaticMethod, the compiler will only look at the declared type of the reference, and use that declared type to determine, at compile time, which method to call. So it will call the static method of Animal! The same thing applies for variables, either instance or static. This is why you should absolutely avoid hiding fields and be VERY careful when calling static methods to invoke them through the class name rather than the object.



## PRACTICE TIME

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}

public class ClassB extends ClassA {
    public static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

1. Which method *overrides* a method in the superclass?
2. Which method *hides* a method in the superclass?
3. What do the other methods do?

Here are some practice questions.

## ANSWER

1. `methodTwo.`

2. `methodFour.`

3. They cause compile-time errors.
   `methodOne: “This static method cannot hide the instance method from ClassA”.`
   `methodThree: “This instance method cannot override the static method from ClassA”.`

methodTwo, being instance will override the one in the superclass. methodFour, being static will hide the one in the superclass. They cause compile-time errors.
methodOne: “This static method cannot hide the instance method from ClassA”.
methodThree: “This instance method cannot override the static method from ClassA”.

## `final` keyword

- Variable: only assigned once in its declaration or in constructor – its value cannot be changed after initialization.
    - E.g., `static final int PI = 3.14;`
- Method: cannot be overridden by subclass.
- Class: cannot be extended.

The last thing I want to talk about is the final keyword which we have seen so far for constant variables. If you declare a method as final, then you force any subclass to not be able to override it. And if you declare a class as final, then no class can extend it.

## Lecture 5: Inheritance

- Inheritance

And that's all for today. Feel free to add a main method to your Cat, Dog, Animal classes and practice with creating objects both with the Subclass obj = new Subclass(); and SuperClass obj = new Subclass(); format.

## Readings:

- Inheritance: https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html

## Code

- Lecture 5 code

## Worksheet

- Lecture 5 worksheet