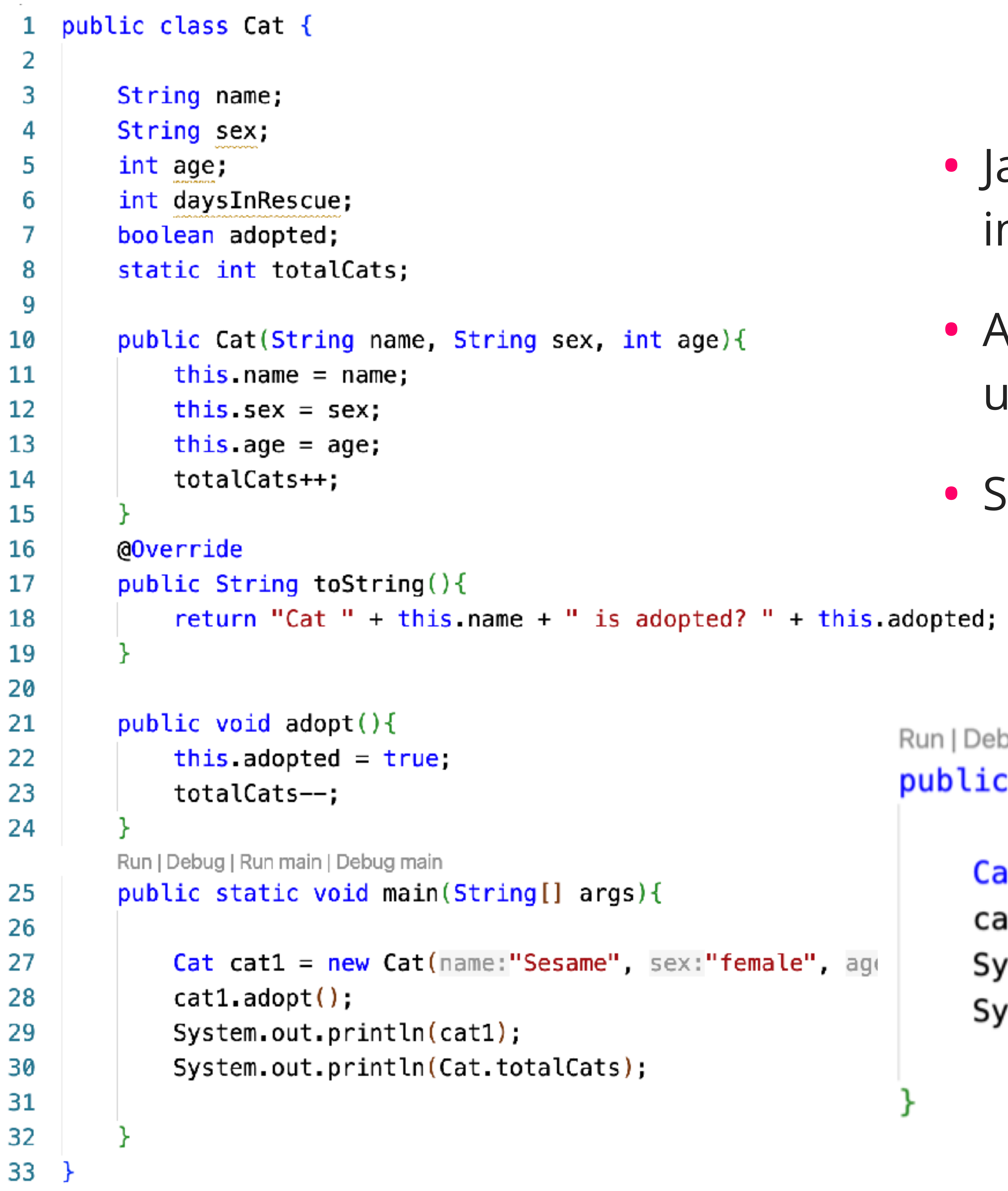
CS62 Class 3: Encapsulation & Inheritance

Java Fundamentals

Last time review

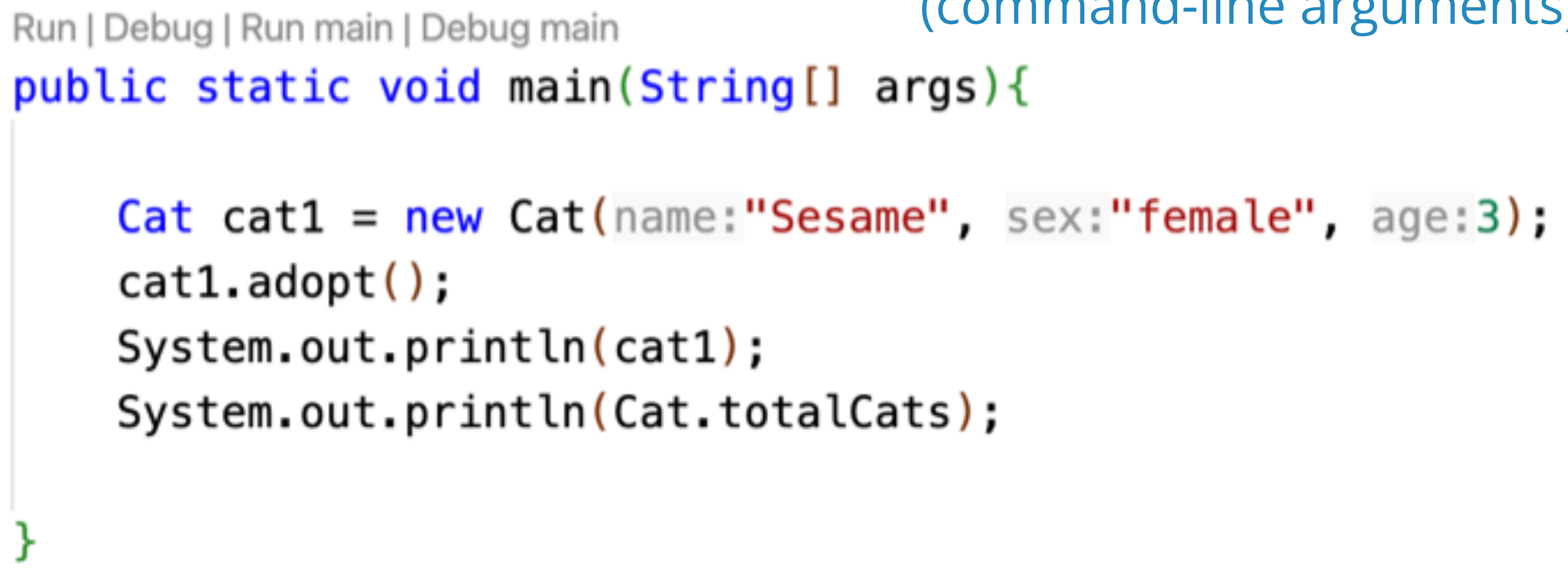
- Java is an object oriented language: constructors, instances, methods
- A method can be declared static iff it does **not** use any instance variables
- Static variables are shared across class instances



```java
public class Cat {

    String name;
    String sex;
    int age;
    int daysInRescue;
    boolean adopted;
    static int totalCats;

    public Cat(String name, String sex, int age){
        this.name = name;
        this.sex = sex;
        this.age = age;
        totalCats++;
    }
    @Override
    public String toString(){
        return "Cat " + this.name + " is adopted? " + this.adopted;
    }

    public void adopt(){
        this.adopted = true;
        totalCats--;
    }
    Run | Debug | Run main | Debug main
    public static void main(String[] args){

        Cat cat1 = new Cat(name:"Sesame", sex:"female", ag
        cat1.adopt();
        System.out.println(cat1);
        System.out.println(Cat.totalCats);

    }
}
```

void = doesn't return anything

String[] args = an array of Strings (command-line arguments)



```java
Run | Debug | Run main | Debug main
public static void main(String[] args){

    Cat cat1 = new Cat(name:"Sesame", sex:"female", age:3);
    cat1.adopt();
    System.out.println(cat1);
    System.out.println(Cat.totalCats);

}
```

Lecture 3 agenda

- Encapsulation, access keywords & data hiding: `public` vs `private` keywords
- Inheritance
- Polymorphism

Encapsulation

Data hiding (aka encapsulation)

- Data hiding is a core concept in Object-Oriented Programming.
- We encapsulate data and related methods in one class and we restrict who can see and modify data.
    - For example, FERPA protects the privacy of students so the Registrar cannot share their academic record freely, even if it's their parents who request it.
- Java uses access modifiers to set the access level for classes, variables, methods and constructors.

Access Modifiers: public, private, default, protected

- You are already familiar with the public keyword. E.g., `public class PomonaStudent`.
- For classes, you can either use **public** or ***default***:
  - **public: The class is accessible by any other class**. E.g.,
    - `public class PomonaStudent`
  - ***default*: The class is only accessible by classes in the same package** (think of it as in the same folder. More soon). This is used when you don't specify a modifier. E.g.,
    - `class PomonaStudent`
- For variables, methods, and constructors, you can use any of the following:
  - `public`: the code is accessible by any other class
  - `private`: The code is only accessible within the declared class
  - *default*: The code is only accessible in the same package. This is used when you don't specify a modifier
  - `protected`: The code is accessible in the same package and subclasses (more later).

Package

- A grouping of related classes that provides access protection and name space management. E.g.,
  - `java.lang` and `java.util` for fundamental classes or `java.io` for classes related to reading input and writing output.
- Packages correspond to folders/directories.
- Lower-case names. E.g.,
  - `package registrar;`
  - at top of file and file has to be within registrar folder
- You may also import built-in Java packages, eg `import java.util.*;` for including all classes
- or `import java.util.Arrays;` for more specific access (Arrays class).

https://docs.oracle.com/javase/tutorial/java/package/packages.html

Data Hiding

- To follow the concept of data hiding, we prefer to define instance variables as `private`.

- We provide more lax (i.e. `default`, `protected`, or `public`) getter and setter methods to access and update the value of a `private` variable.

Demo: PomonaStudent, ScrippsStudent, StudentLauncher

```
package registrar;

public class PomonaStudent {

    private String name;
    private String email;
    private int id;
    private int yearEntered;
    private String academicStanding;
    private boolean graduated;
    private static int studentCounter;


    int getYearEntered() { //default access
        return yearEntered;
    }

    void setYearEntered(int yearEntered) {
        this.yearEntered = yearEntered;
    }

    //private - scripps should not be able to graduate pomona students
    private static void graduateAllStudents(){
        studentCounter = 0;
     }

    public PomonaStudent(String name){
        this.name = name;
        studentCounter++;
    }

    public static PomonaStudent olderStudent(PomonaStudent p1, PomonaStudent p2) {
        if (p1.yearEntered < p2.yearEntered) {
            //if they entered an earlier year, they are an older (in class ranking) student
            return p1;
        }
        return p2;
    }

    public String toString(){
        return "Pomona Student \n Name: " + name + "\n email: " + email + "\n id: " + id;
    }

}
```

we have moved the file to the registrar/ folder and declared `package registrar`

all the instance variables have been declared `private`

getters and setters are `default` access (any code in the package can use them)

Worksheet time!

Given the PomonaStudent and ScrippsStudent classes we just showed in class, there are 4 incorrect lines of code in the StudentLauncher class below. Identify the lines and reason about a workaround. (Please refer to the code link on the class website for reference for the classes.)

```java
package Registrar;
public class StudentLauncher {

    public static void main(String[] args) {
        PomonaStudent student1 = new PomonaStudent("cecil");
        student1.setYearEntered(2022);
        System.out.println(student1.getYearEntered());
        System.out.println(student1.yearEntered);
        PomonaStudent.graduateAllStudents();

        ScrippsStudent student2 = new ScrippsStudent("La Semeuse");
        student2.setYearEntered(1889);
        System.out.println(student1);
        System.out.println(student2);
        System.out.println(PomonaStudent.olderStudent(student1,
                           student2));
    }
}
```

Sometimes, we want a .java file to just be the class definition (no main), and we'll have a separate "launcher" .java file with a main method.

What's wrong with StudentLauncher?

Worksheet answers

Given the PomonaStudent and ScrippsStudent classes we just showed in class, there are 4 incorrect lines of code in the StudentLauncher class below. Identify the lines and reason about a workaround. (Please refer to the code link on the class website for reference for the classes.)

```java
package Registrar;
public class StudentLauncher {

    public static void main(String[] args) {
        PomonaStudent student1 = new PomonaStudent("cecil");
        student1.setYearEntered(2022);
        System.out.println(student1.getYearEntered());
        System.out.println(student1.yearEntered);
        PomonaStudent.graduateAllStudents();

        ScrippsStudent student2 = new ScrippsStudent("La Semeuse");
        student2.setYearEntered(1889);
        System.out.println(student1);
        System.out.println(student2);
        System.out.println(PomonaStudent.olderStudent(student1,
                                   student2));
    }
}
```

registrar (lower case)

yearEntered is a private variable, need getYearEntered()

graduateAllStudents is a private method, cannot use here

type error - needs 2 PomonaStudents, but student2 is ScrippsStudent

Inheritance

The last example

- Why is olderStudent a method that takes only parameters of type PomonaStudent only? It would be nice to compare both PomonaStudents and ScrippsStudents without having to write 3 different signatures
  - `olderStudent(PomonaStudent p1, PomonaStudent p2)`
  - `olderStudent(ScrippsStudent p1, ScrippsStudent p2)`
  - `olderStudent(PomonaStudent p1, ScrippsStudent p2)`
- What if we made a general Student class, which could define commonalities shared between PomonaStudents and ScrippsStudents? Then we could write data specific to each school (e.g., ID1 course for Pomona students) in the more specialized classes.

Inheritance conceptual overview

- Classes can be parent/child classes of each other (subclasses)

class Student
getName()
class PomonaStudent
getLanguageReqCompleted()
class ScrippsStudent
getGWSReqCompleted()
class FourthYearPomonaStudent
getThesisAdvisor()
class FirstYearPomonaStudent
getID1course()

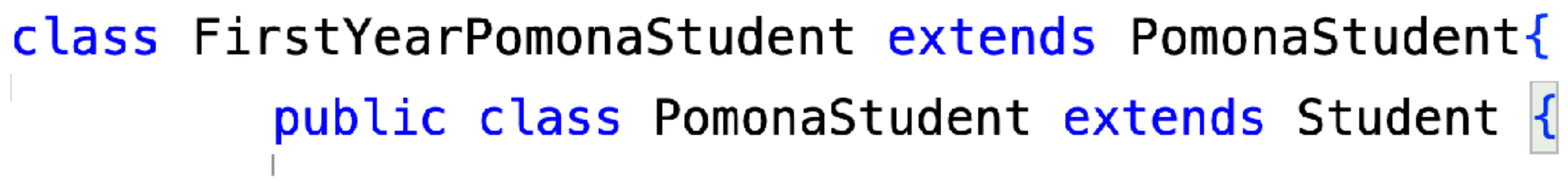
Inheritance

- 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.

- 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.

- Java allows multilevel inheritance: A class can extend a class which extends a class etc.



```
class FirstYearPomonaStudent extends PomonaStudent{
```

```
public class PomonaStudent extends Student {
```

Inheritance & encapsulation

- The subclass inherits all the `public` and `protected` variables and methods.
  - Not the `private` ones, although it can access them with appropriate getters and setters.
- The inherited variables can be used directly, just like any other variables.
- 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 have already done that! (How?)

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 String toString()`
  - Returns string representation of object – default is hexadecimal hash of memory location.
  - We've overrode this!
- `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 int hashCode()`
  - Unique identifier defined so that if `a.equals(b)` then `a, b` have same hash code (more later).

use the `super` keyword to access the parent

- Refers to the direct parent of the subclass.
  - E.g., `super` in `FirstYearPomonaStudent` refers to `PomonaStudent`
- `super.instanceMethod()`: for overridden methods.
  - What is an overridden method? If FirstYearPomonaStudent has a method that's the same name as a method in PomonaStudent, but we want to call the PomonaStudent one instead, we need to use `super`.
- `super(args)`: to call the constructor of the super class. Should be called in the first line of the subclass's constructor.

Code example

moved shared variables/methods to Student class

created a general olderStudent method

```java
class Student {

    private String name;
    private String email;
    private int id;
    private String major;
    private int yearEntered;

    private static int studentCounter;

    protected Student(String name, String email, int id){
        this.name = name;
        this.email = email;
        this.id = id;
        studentCounter++;
        major = "Undeclared";
    }
...
    protected static int getStudentCounter(){
        return studentCounter;
    }

    protected static void setStudentCounter(int students){
        studentCounter = students;
    }

    protected int getMaxCredits(){
        return 4;
    }

    public String toString(){
        return "Student Info - Name: " + name + "\nemail: " + email + "\nid: " + id + "\n";
    }

    public static Student olderStudent(Student p1, Student p2) {
        if (p1.getYearEntered() < p2.getYearEntered()) {
            //if they entered an earlier year, they are an older (in class ranking) student
            return p1;
        }
        return p2;
    }

}
```

```java
public class PomonaStudent extends Student {

    private boolean languageReqCompleted;

    private static int pomonaStudentCounter;

    protected PomonaStudent(String name, String email, int id){
        super(name, email, id);
        pomonaStudentCounter++;
    }

    protected void completeLanguageReq() {
        languageReqCompleted = true;
    }

    protected boolean getLanguageReqCompleted() {
        return languageReqCompleted;
    }

    public String toString(){
        return "Pomona " + super.toString();
    }

    protected static void graduateAllStudents(){
        Student.setStudentCounter(Student.getStudentCounter()
        - pomonaStudentCounter);
        pomonaStudentCounter = 0;
    }
}
```

difference between subclasses is language vs GWS requirement

calls Student.toString()

each class has their own graduateAllStudents and counter, which also changes the overall Student counter

```java
public class ScrippsStudent extends Student {

    private boolean gwsReqCompleted;

    private static int scrippsStudentCounter;

    protected ScrippsStudent(String name, String email, int id){
        super(name, email, id);
        scrippsStudentCounter++;
    }

    protected void completeGWSReq() {
        gwsReqCompleted = true;
    }

    protected boolean getGWSReqCompleted() {
        return gwsReqCompleted;
    }

    public String toString(){
        return "Scripps " + super.toString();
    }

    protected static void graduateAllStudents(){
        Student.setStudentCounter(Student.getStudentCounter()
        - scrippsStudentCounter);
        scrippsStudentCounter = 0;
    }
}
```

Q: Why do we need super? Why can't we write Student.toString() directly?

A: the class name syntax is reserved for static methods! Java will think you're trying to call a static method, instead of the instance method of the parent class.

```java
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++;
    }

    protected String getId1(){
        return id1;
    }

    protected void setID1(String id1){
        this.id1 = id1;
    }

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

}
```

Between First and Fourth year Pomona student, difference is having new ID1 versus thesis instance variables

```java
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;
        fourthYearCounter++;
    }

    protected String getThesisTitle(){
        return thesisTitle;
    }

    protected void setThesisTitle(String thesisTitle){
        this.thesisTitle = thesisTitle;
    }

    @Override
    protected int getMaxCredits(){
        return 6;
    }

    public String toString(){
        return super.toString() + "Fourth-Year Student Writing
Thesis on: " + thesisTitle;
    }
}
```

Also, this @overrides the getMaxCredits() in Student.java, which was only 4

```java
public class App {

    public static void main(String[] args) {

        FirstYearPomonaStudent student1 = new FirstYearPomonaStudent("daniel", "daniel@pomona.edu", 1, "War and Peace");
        FirstYearPomonaStudent student2 = new FirstYearPomonaStudent("wentao", "wentao@pomona.edu", 2, "Everday Chemistry");
        ScrippsStudent student3 = new ScrippsStudent("stacy", "stacy@scripps.edu", 3);
        ScrippsStudent student4 = new ScrippsStudent("abram", "abram@scripps.edu", 4);
        FourthYearPomonaStudent student5 = new FourthYearPomonaStudent("millie", "millie@pomona.edu", 5, "Power and its
        Effects on Software Engineering");
        FirstYearPomonaStudent[] firstYears = {student1, student2};

        for (FirstYearPomonaStudent firstYear : firstYears) {
            System.out.println(firstYear);
            System.out.println("---");
        }
```

An array with the most specific type (FirstYearPomonaStudent)

```java
        //can use more general parent class
        Student[] students = new Student[5];
        students[0] = student1;
        students[1] = student2;
        students[2] = student3;
        students[3] = student4;
        students[4] = student5;
        for (Student student : students) {
            System.out.println(student);
            System.out.println("---");
        }
```

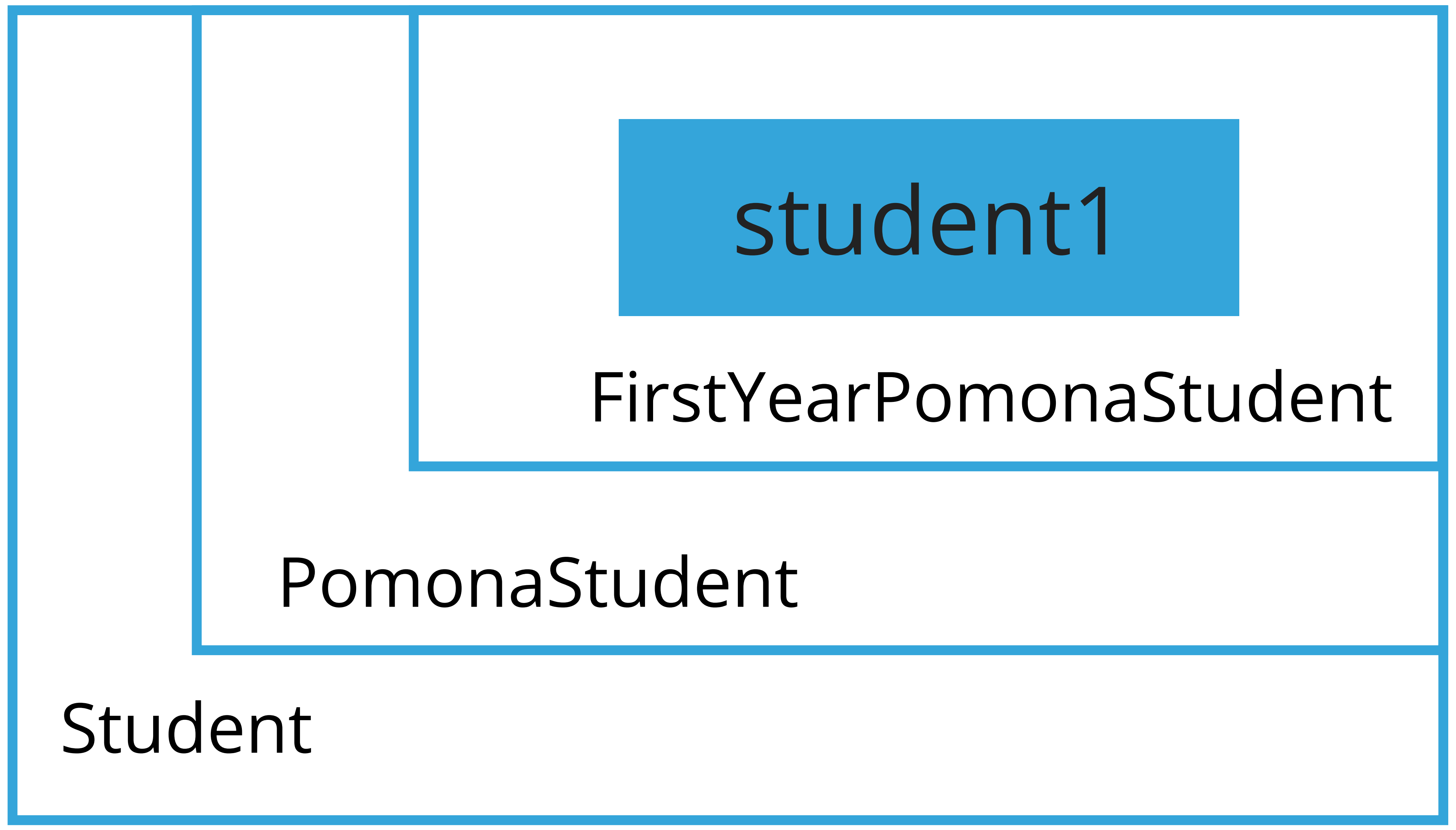
An array with the most general type (Student)

```java
        //checking graduating students
        System.out.println(Student.getStudentCounter()); //should be 5
        PomonaStudent.graduateAllStudents();
        System.out.println(Student.getStudentCounter()); //now should be 2
        ScrippsStudent.graduateAllStudents();
        System.out.println(Student.getStudentCounter()); //now should be 0

    }
}
```

Polymorphism

Polymorphism

- Polymorphism means one object can take many forms: they can use instance variables and methods (public/protected/default) from many classes.
  - FirstYearPomonaStudents are still PomonaStudents are still Students are still Objects



Polymorphism

```
FirstYearPomonaStudent student1 = new FirstYearPomonaStudent("daniel", "daniel@pomona.edu", 1,
"War and Peace");
FirstYearPomonaStudent student2 = new FirstYearPomonaStudent("wentao", "wentao@pomona.edu", 2,
"Everday Chemistry");
ScrippsStudent student3 = new ScrippsStudent("stacy", "stacy@scripps.edu", 3);
ScrippsStudent student4 = new ScrippsStudent("abram", "abram@scripps.edu", 4);
FourthYearPomonaStudent student5 = new FourthYearPomonaStudent("millie", "millie@pomona.edu", 5,
"Power and its Effects on Software Engineering");

//can use more general parent class
Student[] students = new Student[5];
students[0] = student1;
students[1] = student2;
students[2] = student3;
students[3] = student4;
students[4] = student5;
for (Student student : students) {
    System.out.println(student);
    System.out.println("---");
}
```

Polymorphism

For flexibly changing objects between child classes, use this syntax:

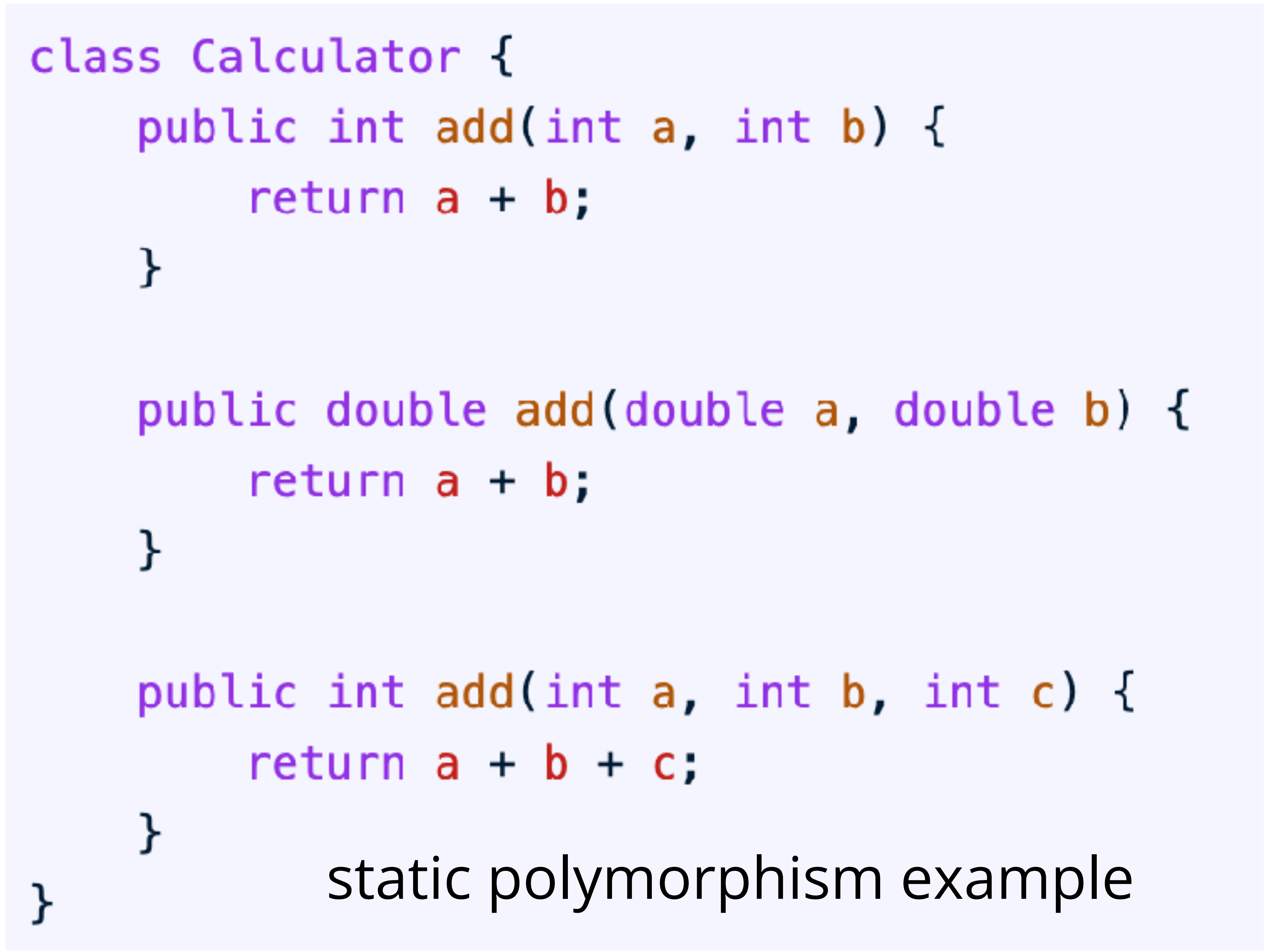
```
ParentClass obj = new ChildClass();
```

```
Student student1 = new FirstYearPomonaStudent(“frank”);
```
✅

```
FirstYearPomonaStudent student1 = new Student(“frank”);
```
❌

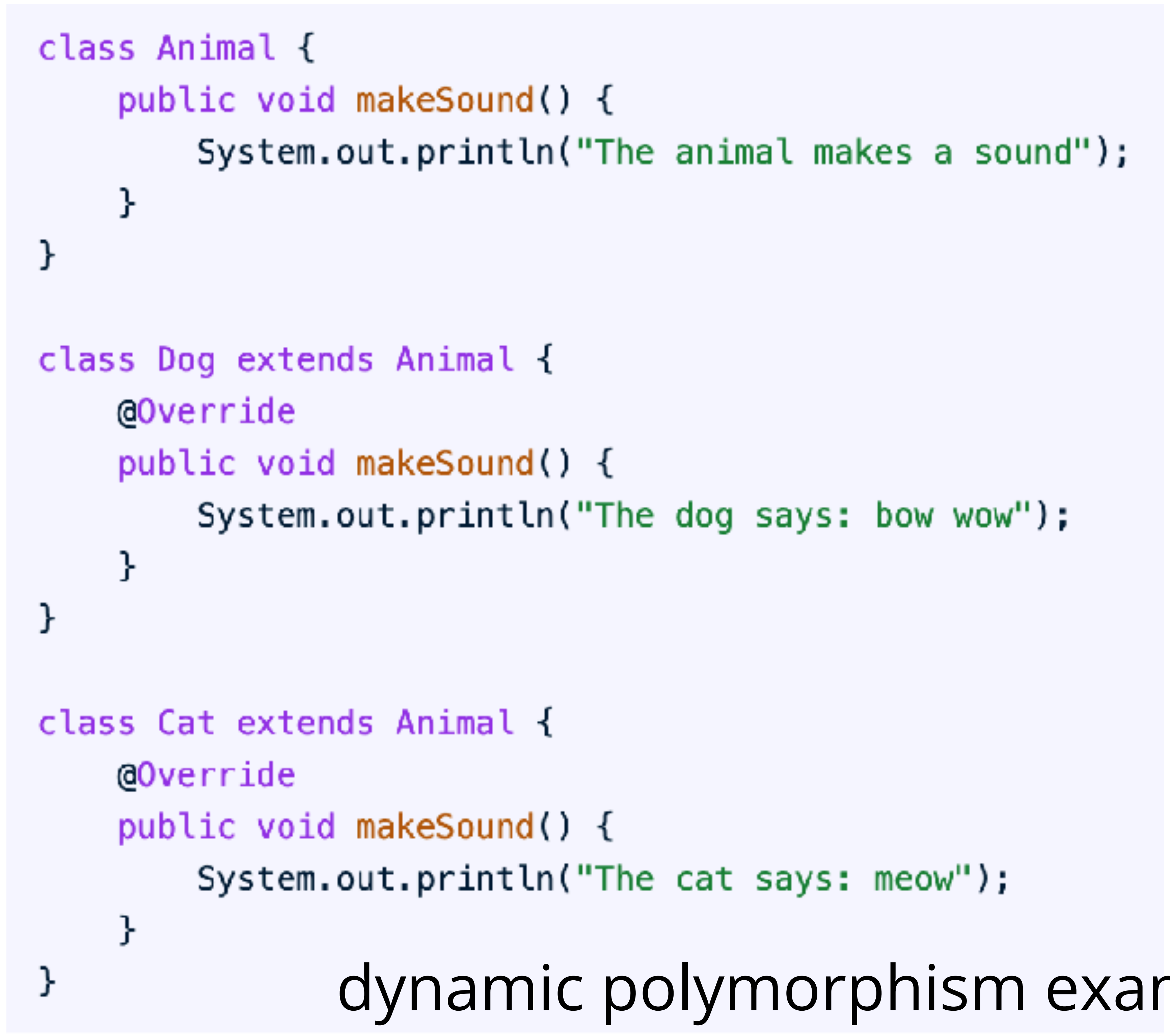
Overriding, dynamic vs static polymorphism

- Overriding: **Instance methods** in child classes **override** the instance methods in the parent classes (like .getMaxCredits()).
- This is called dynamic polymorphism, since it happens at runtime.
- In contrast to static polymorphism, which happens when we overload methods (such as having multiple constructors).



```
class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

static polymorphism example



```
class Animal {
    public void makeSound() {
        System.out.println("The animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The dog says: bow wow");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("The cat says: meow");
    }
}
```

dynamic polymorphism example

Method hiding

- Method hiding occurs when a subclass defines a **static method** with the same signature as a static method in its superclass.
- Unlike instance methods, which can be overridden, static methods are resolved at compile time based on the **class type** (the type on the left side), not the object type.
- Same thing happens with all variables: both static and instance.

Remember: static methods only, but all variables

```java
class Parent {
    public static void display() {
        System.out.println("Parent method");
    }
}

class Child extends Parent {
    public static void display() {
        System.out.println("Child method");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent p = new Parent();
        p.display(); // Output: Parent method

        Child c = new Child();
        c.display(); // Output: Child method

        Parent p2 = new Child();
        p2.display(); // Output: Parent method (due to method hiding)
    }
}
```

The important thing is this type here

Example: `Animal`

```java
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");
    }
}
```

Example: Cat

```java
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");
    }
}
```

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 we expected (hopefully).

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  Used the Animal method because of the Animal type
The instance method in Cat  Used the Cat method because it was overriden
2          Used the Animal instance variable because of the Animal type
Animal     Used the Animal static variable because of the Animal type
```

Summary

- If something’s type matches its class (e.g. Parent = new Parent(), Child = new Child()), then just use the variables/methods (both static and instance) in that class
- However, tricky things happen when you have Parent = new Child()
  - Use Parent’s **static methods** and **all** variables (instance + static)
  - Use Child’s **instance methods** and **no** variables

Worksheet 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?

Worksheet answers

```
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?
  - methodTwo
- 2. Which method *hides* a method in the superclass?
  - methodFour
- 3. What do the other methods do?
  - 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”.

File I/O / Java Misc

There’s a few more things to know about Java

- Now we’ve covered most of Java’s quirks, but there are some good-to-know syntax (like, how do you read user input from the console?)
- Fall semester is 1 week shorter than spring semester, so I’ve moved this lecture into **pre-reading** for the lab. **Please read it before lab** (or when you need to use it in your assignments). (It’s not particularly exciting to teach…)
    - There will be a quiz question on these concepts (nothing that requires you to memorize code, like I/O streams)
    - Note for quizzes: closed note, but can retake

I/O streams

In Python, it was open()… read()… write()…

- Input stream: a stream from which a program reads its input data
- Output stream: a stream to which a program writes its output data
- Error stream: output stream used to output error messages or diagnostics
- Stream sources and destinations include disk files, keyboard, peripherals, memory arrays, other programs, etc.
- Data stored in variables, objects and data structures are temporary and lost when the program terminates. Streams allow us to save them in files, e.g., on disk or flash drive or even a CD (!)
- Streams can support different kinds of data: bytes, characters, objects, etc.

https://docs.oracle.com/javase/tutorial/essential/io/streams.html

Files

- Every file is placed in a directory in the file system.
- Absolute file name: the file name with its complete path and drive letter. E.g.,
  - On Windows: `C:\jli\somefile.txt`
  - On Mac/Unix: `/~/jli/somefile.txt`
- Caution: directory separator in Windows is \, which is A special character in Java. Should be “\\” instead.
- `File` class: contains methods for obtaining file properties, renaming, and deleting files. Not for reading/writing!

Writing data to a text file

- `PrintWriter output = new PrintWriter(new File("filename"));`
- If the file already exists, it will overwrite it. Otherwise, new file will be created.
- Invoking the constructor may throw an IOException so we will need to follow the catch or specify rule.
- `output.print` and `output.println` work with Strings, and primitives.
- Always close a stream!

Writing data to a text file

```
import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;

public class WriteData {
    public static void main(String[] args) {

        PrintWriter output = null;
        try {
            output = new PrintWriter(new File("addresses.txt"));
            // Write formatted output to the file
            output.print("Alexandra Papoutsaki ");
            output.println(222);
            output.print("Jingyi Li ");
            output.println(111);

        } catch (IOException e) {
            System.err.println(e.getMessage());
        } finally {
            if (output != null)
                output.close();
        }
    }
}
```

need to import relevant classes

call .print or .println to write to file

catch IOException for any errors

.close() the I/O stream

https://liveexample.pearsoncmg.com/html/WriteData.html

Reading data

- `java.util.Scanner` reads Strings and primitives and breaks input into tokens, denoted by whitespaces.

You'll see this in HW2

- **To read from keyboard:** `Scanner inputStream = new Scanner(System.in);`
  - `String input = inputStream.nextLine();`
  - `input` is a String. If you want to convert it into a number, you will need to use the wrapper class of the primitive you want, e.g., `Integer.parseInt(input);`
- To read from file: `Scanner inputStream = new Scanner(new File("filename"));`
- Need to close stream as before.
- `inputStream.hasNext()` tells us if there are more tokens in the stream. `inputStream.next()` returns one token at a time.
  - Variations of next are `nextLine()`, `nextByte()`, `nextShort()`, etc.

Reading data from a text file

```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class ReadData {
   public static void main(String[] args) {

      Scanner input = null;
      // Create a Scanner for the file
      try {
         input = new Scanner(new File("addresses.txt"));

         // Read data from a file
         while (input.hasNext()) {
            String firstName = input.next();
            String lastName = input.next();
            int room = input.nextInt();
            System.out.println(firstName + " " + lastName + " " + room);
         }
      } catch (IOException e) {
         System.err.println(e.getMessage());
      } finally {
         if (input != null)
            input.close();
      }
   }
}
```

same try...catch...finally structure

use Scanner class

use a while loop to check if file still has lines

.next() is space separated (if you want the whole line, call .nextLine())

close the file

Full example/reference:

https://github.com/pomonacs622025sp/code/blob/main/Lecture3/FileIOExample.java

https://liveexample.pearsoncmg.com/html/ReadData.html

Reading data with a buffered reader

- `import java.io.FileReader;`
- `import java.io.BufferedReader;`

```
FileReader fr = new FileReader(“fileToRead.txt”);

BufferedReader br = new BufferedReader(fr);

String line = br.readLine();

while ((line!= null) {

    //do something

    line = br.readLine();

}
```

a BufferedReader object takes a FileReader object as input.

the .readLine() method will return null when the file has no more lines to read, so we can write a while loop

You’ll see this in HW3

Lecture 3 wrap-up

- Data and methods can be declare public, private, default, or protected
- A class can inherit from another class
- TODO: HW2 released, due next Tuesday
- Lab requires some pre-reading of miscellaneous Java syntax. It will be a quiz question!
- Your first quiz will be in lab tonight. 5-10 minutes; don't stress, we have regrades. Covers everything we've seen so far (including today)

Resources

- Inheritance: https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdev/Java-overview.html#GUID-5C62367E-3197-4E67-A38E-39CE04C7B795
- Polymorphism: https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html
- See after this slide for more practice problems

Optional problem #1

Recall your Cat class. You also made a Dog class for the animal shelter, but realized there are lots of commonalities – name, sex, age, daysInRescue. Let's make a parent class Animal that both Dog and Cat can extend. From your research, people who adopt cats care about their furType (short, long, etc.) and people who adopt dogs care about their breed (Corgi, Golden Retriever, etc.). Write 3 classes to represent this information. Be sure to:

- Put all the classes in an appropriate package
- Choose the right access modifiers for your fields and methods
- Have getter and setter methods for your instance variables
- Have a constructor (that takes all the relevant parameters) and a counter variable for each class
- Have a toString() method for each class, with Dog and Cat calling the Animal's toString() before adding their own information.

(There's no starter code for this problem: practice remembering the syntax by yourself!)

- Solution: https://github.com/pomonacs622025sp/code/tree/main/Lecture4/animalShelter

Optional problem #2

Change graduateAllStudents() in the PomonaStudent class to only graduate FourthYearPomonaStudents instead of all students.

Write a method `protected FourthYearPomonaStudent firstToFourthYear(FirstYearPomonaStudent p1, String thesisTitle)` that transitions p1 from a first year to a fourth year. Skipped 2 grades!

(Answer: exercise to the reader! Write test code to see!)