Announcements

- Last week's lab
- Midterm exam 1:
  - September 26, 8:00 – 9:00pm
  - Two rooms EE129 & FRNY G140
  - You will be assigned a room
  - Coverage: upto Week 5.
  - Closed book/notes. Can bring one sheet.
- Piazza posts
  - don't be intimidated by topics outside of class.

User Defined Classes

CS 18000
Sunil Prabhakar
Department of Computer Science
Purdue University

Objectives

In this module, we will study how to create user defined classes including:

- Creating objects from user defined classes
- Adding data members and methods to classes
- Class and object methods
- Argument passing for methods in Java
- Scope of variables

PURDUE UNIVERSITY

Problem

- *Create a program that reads in details about your course staff and then prints them out in a neat table.*
  - *Instructor (me), Course Coordinator, and TA.*
  - *For each individual, we need to record the following (String) values:*
    - *First Name, Last Name*
    - *Email*
    - *Office*

Sample output

Instructor:
Sunil Prabhakar
Office: LWSN 3144F
Email: sunil@purdue.edu
Course Coordinator:
Lorenzo Martino
Office: HAAS 144
Email: lmartino@purdue.edu
Teaching Assistant:
David Runyan
Office: LWSN B116B
Email: drunyan@purdue.edu

Solution

- Since we are creating multiple versions of the same data and performing the same operations, it would be very helpful to have a class for saving each person's data and printing it out neatly.
- No such standard class exists.
- We will create one: CS180Staff
- Our program will also be a separate class that will use this CS180Staff class.
  - this will be our controller class: CourseStaff

The CS180Staff class

- We want each object of this class to
  - store the data for a single person:
    - these are data members of each object
    - ***firstName, lastName, email, office***
  - input the values for each person
    - this is a behavior for which we define a method
    - ***readDetails();***
  - print out the details of the person neatly
    - this is a behavior for which we define a method
    - ***printNeatly()***;

CS180Staff class

```
import javax.swing.*;

public class CS180Staff {

   private String firstName, lastName, email, office;

   void getDetails(){
       firstName = JOptionPane.showInputDialog(null, "Enter First Name:");
       lastName = JOptionPane.showInputDialog(null, "Enter Last Name:");
       email = JOptionPane.showInputDialog(null, "Enter email:");
       office = JOptionPane.showInputDialog(null, "Enter office:");
   }

   void printNeatly(){
             System.out.println("    " + firstName + " " + lastName);
             System.out.println("    Email: " +  email);
             System.out.println("    Office: " +  office);
   }
}
```

Controller class: CourseStaff

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

        CS180Staff instructor, coordinator,ta;

        instructor = new CS180Staff();
        instructor.getDetails();

        coordinator = new CS180Staff();
        coordinator.getDetails();

        ta = new CS180Staff();
        ta.getDetails();

        System.out.println("Instructor:");
        instructor.printNeatly();

        System.out.println("Coordinator:");
        coordinator.printNeatly();

        System.out.println("Teaching Assistant:");
        ta.printNeatly();
   }
}
```

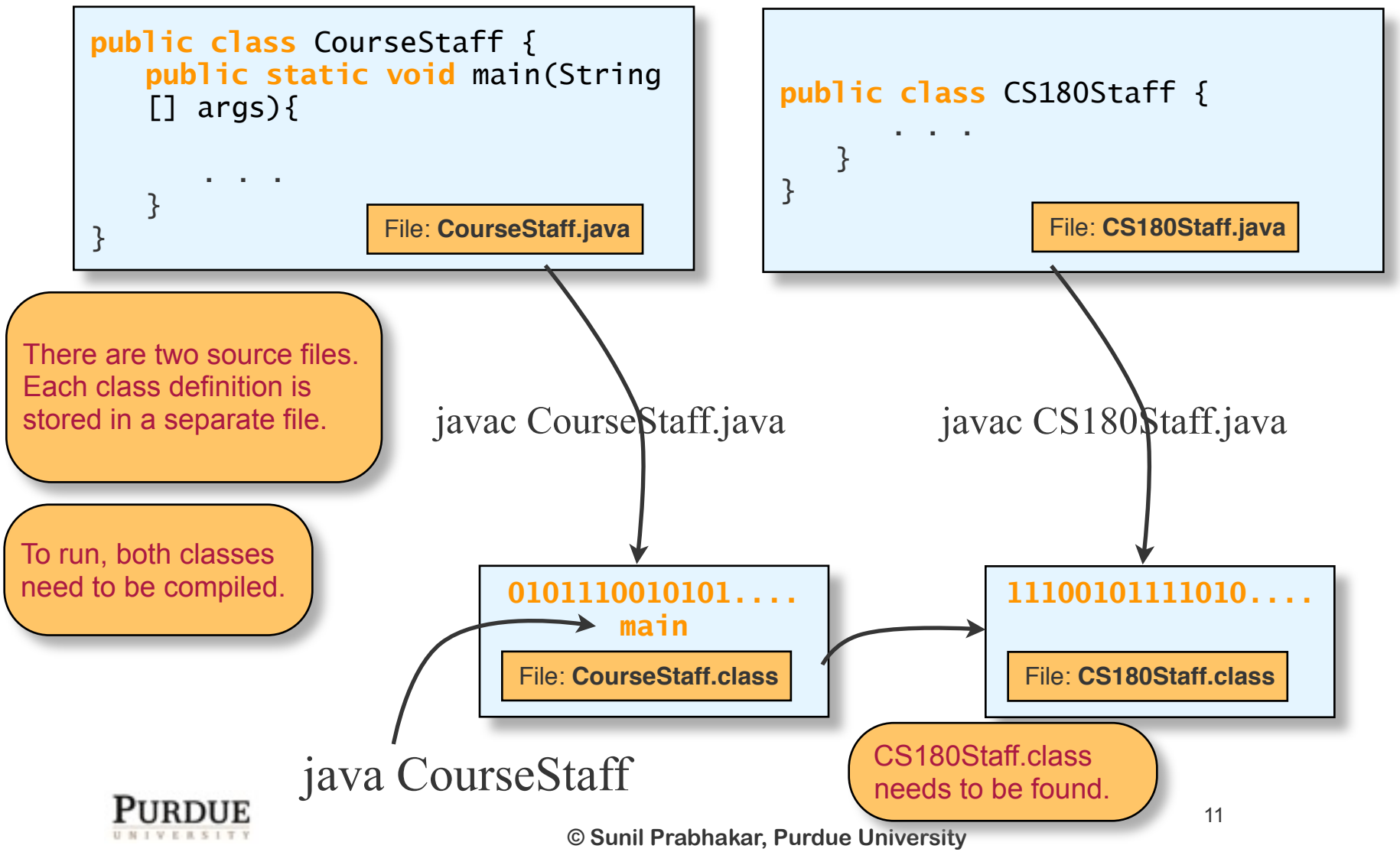
CS180Staff class

```
import javax.swing.*;

public class CS180Staff {

   private  String firstName, lastName, email, office;

   void getDetails(){
       firstName = JOptionPane.showInputDialog(null, "Enter First
       Name:");
       lastName = JOptionPane.showInputDialog(null, "Enter Last Name:");
       email = JOptionPane.showInputDialog(null, "Enter email:");
       office = JOptionPane.showInputDialog(null, "Enter office:");
   }

   void printNeatly(){
            System.out.println("    " + firstName + " " + lastName);
            System.out.println("    Email: " +  email);
            System.out.println("    Office: " +  office);
   }
}
```

Data Members declared outside any method

Methods.

Files and Classes

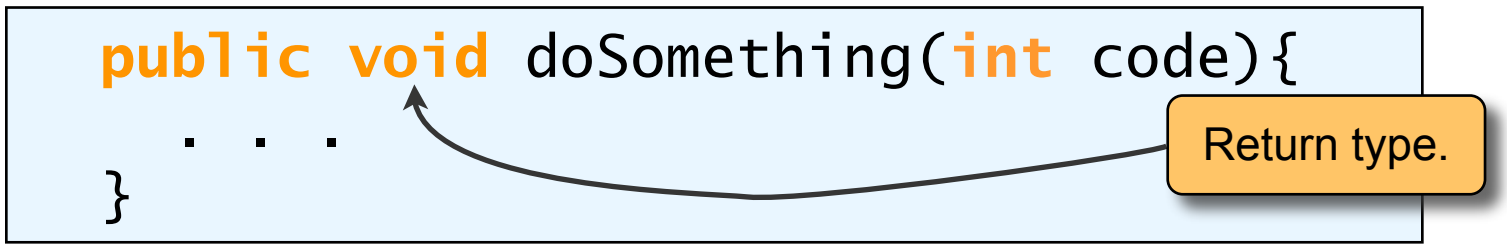


Methods

- Note that we have created two methods for the CS180Staff class.
- Each corresponds to a well-defined piece of work.
- This makes it easy to use methods.
- Similarly, the Math class methods all perform a well-defined operation.
- Any number of methods can be defined for a class.
- Method names should give clues to their function. Typically verbs (e.g, printDetails()).

Adding methods

- Methods for a class are defined as follows:



- The name of the method is preceded by a type indicating the type of value returned by this method when it is done. May be:
  - a primitive type,
  - a Class, or
  - **void** -- indicating that nothing is returned.
- A method may take arguments (e.g., main)

Execution flow

- Our programs begin execution at the first statement in the *main* method.
- Statement are executed in order.
- When a method is called,
  - the execution moves to the first line of that method
  - each statement is executed in order, until
  - we get a **return** statement, or the end of the method
  - then control returns back to the caller.

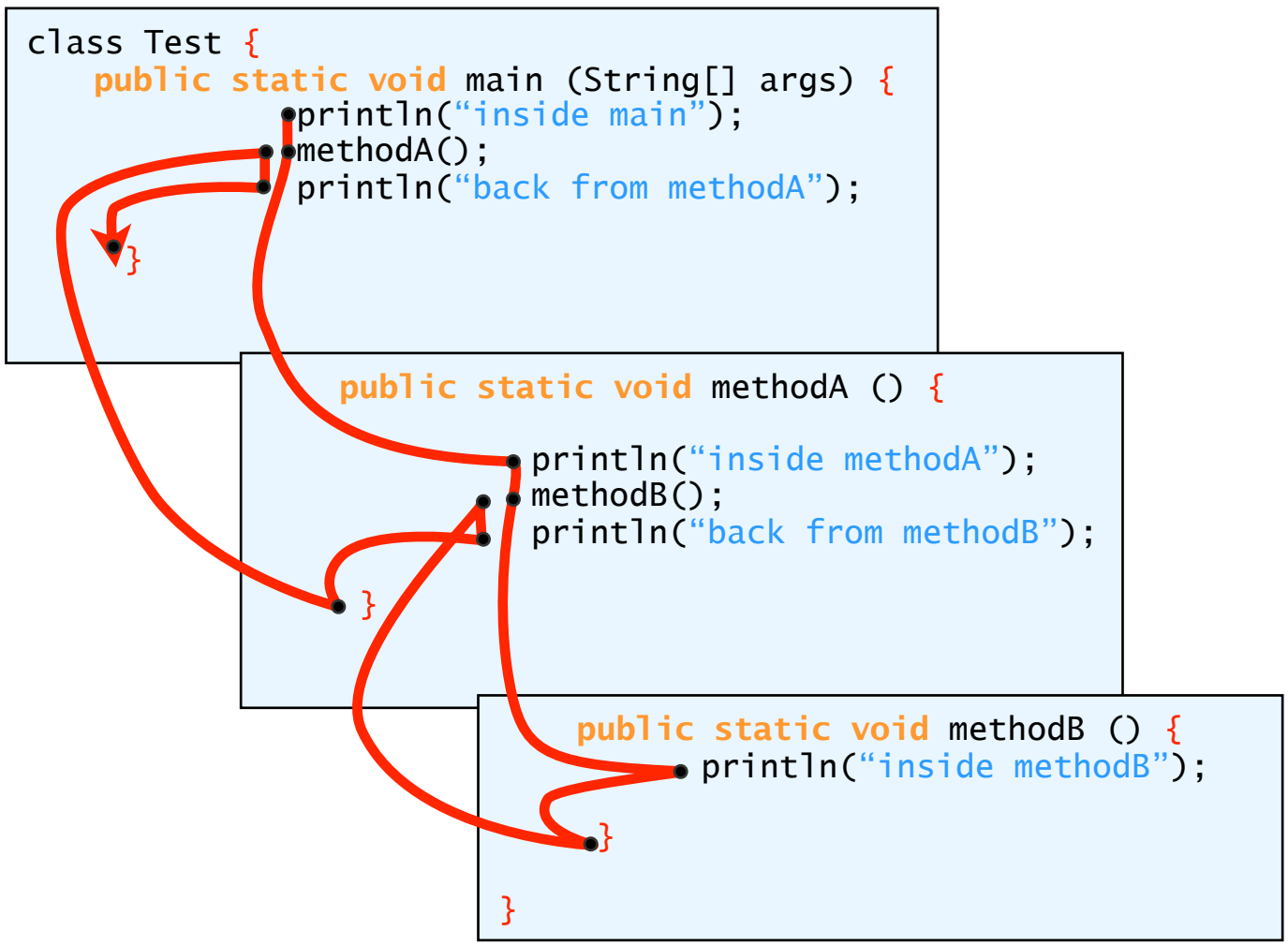
Example flow

```
class Test {
    public static void main (String[] args) {
        println("inside main");
        methodA();
        println("back from methodA");

    }
```

```
    public static void methodA () {

        println("inside methodA");
        methodB();
        println("back from methodB");

    }
```

```
    public static void methodB () {
        println("inside methodB");

    }

}
```

Sample Execution Flow



Adding Data Members

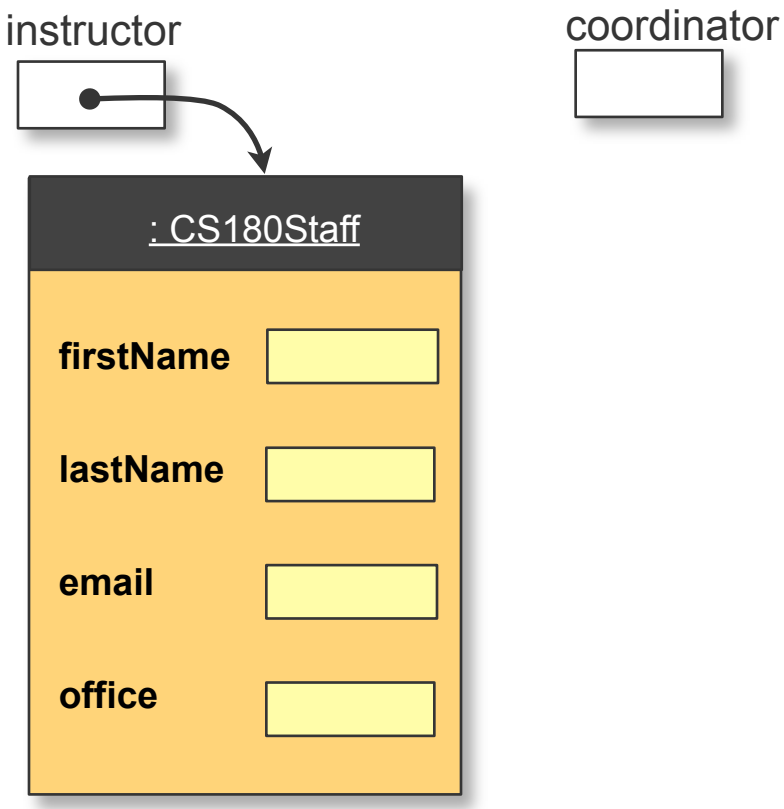
- Data members are declared outside any method. Typically before any method.

```
private      String     firstName;
```

- `private` is a modifier. We will understand it later. It is an alternative to `public` which we have seen before.
  - this is optional -- if left out, it is assumed to be `public`
- Each object of this class will get its own copy of each data member.

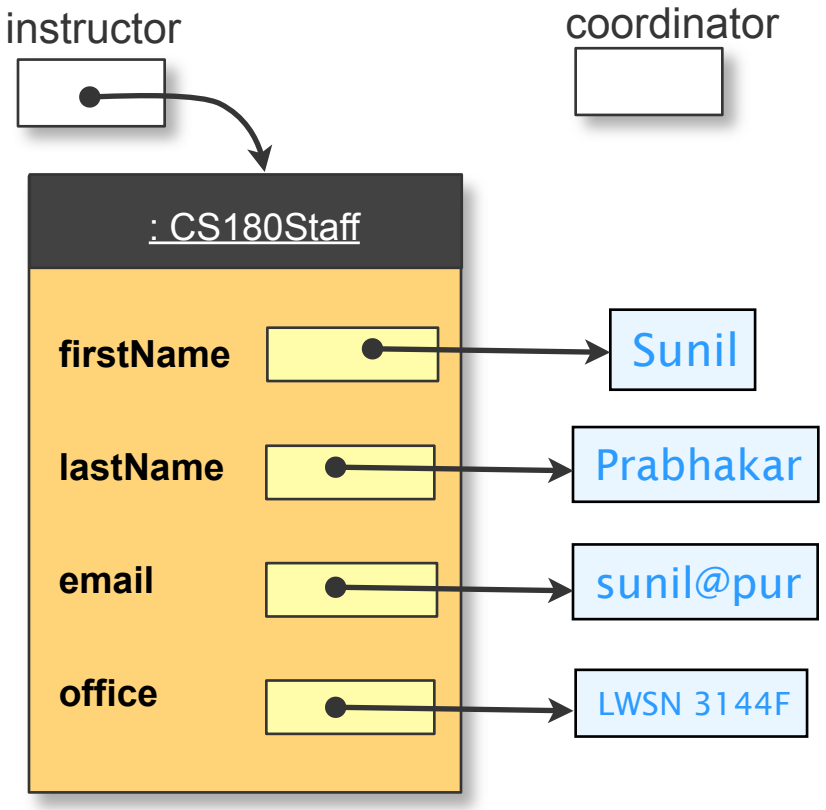
What is happening?

```
CS180Staff instructor, coordinator;

instructor = new CS180Staff();
instructor.getDetails();
```



What is happening?

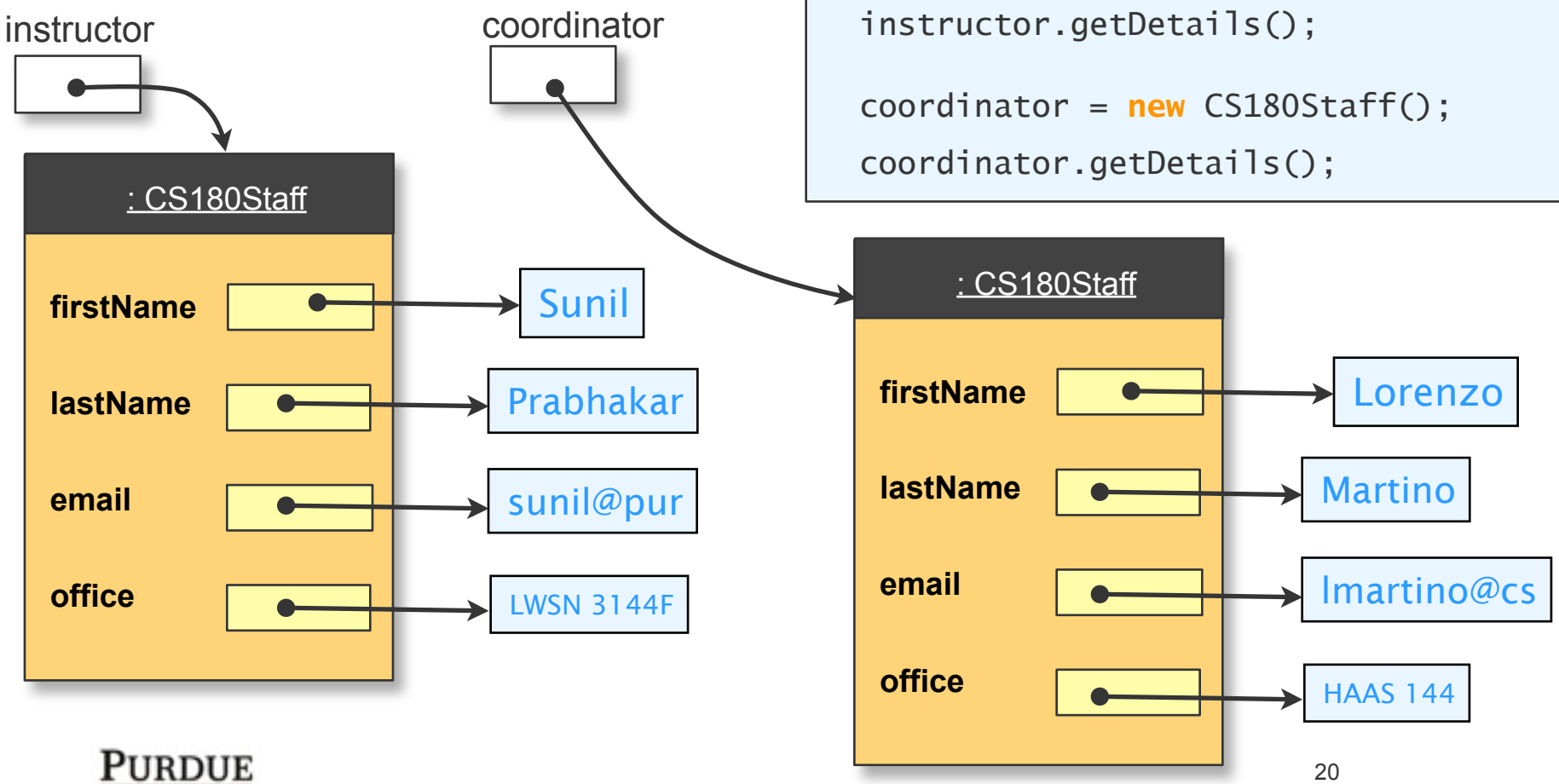
```
CS180Staff instructor, coordinator;

instructor = new CS180Staff();
instructor.getDetails();
```




```
void getDetails(){
  firstName = JOptionPane.showInputDialog
                (null, "Enter First Name:");
  lastName = JOptionPane.showInputDialog
                (null, "Enter Last Name:");
  email = JOptionPane.showInputDialog
                (null, "Enter Email:");
  office = JOptionPane.showInputDialog
                (null, "Enter Office:");
}
```

What is happening?

```
CS180Staff instructor, coordinator;

instructor = new CS180Staff();
instructor.getDetails();

coordinator = new CS180Staff();
coordinator.getDetails();
```



Important point

- Calling **new** CS180Staff creates a new object of this class with its own copy of all data members.
- When a method is called on such an object it modifies only that object's copies of the data members (e.g., firstName).
  - Thus instructor.getDetails() causes only the data members of the object referenced by instructor to be affected, not other CS180Staff objects.

Problem: Course Participants

- *Create a program to input course participants info for Staff (as before), and 2 students, then print it out neatly. Each student*
  - *has a name, gpa (double), and*
  - *is identified by her ID (String)*
  - *a student object must always have a valid ID.*
- *We should be able to read and change the name and gpa of the student at any time, but not the ID.*

The Student class

```
class Student {
  private String name, id;
  private double gpa;

  public void getDetails(){
    name = JOptionPane.showInputDialog(null, "Enter Name:");
    id = JOptionPane.showInputDialog(null, "Enter ID:");
    gpa = 0.0;
  }

  public void printNeatly(){
    System.out.println("    " + name);
    System.out.println("    ID: " +  id);
    System.out.println("    GPA: " +  gpa);
  }

  public void setName(String studentName){
   name = studentName;
  }
  public String getName(){
    return name;
  }
// CONTINUED ...
```

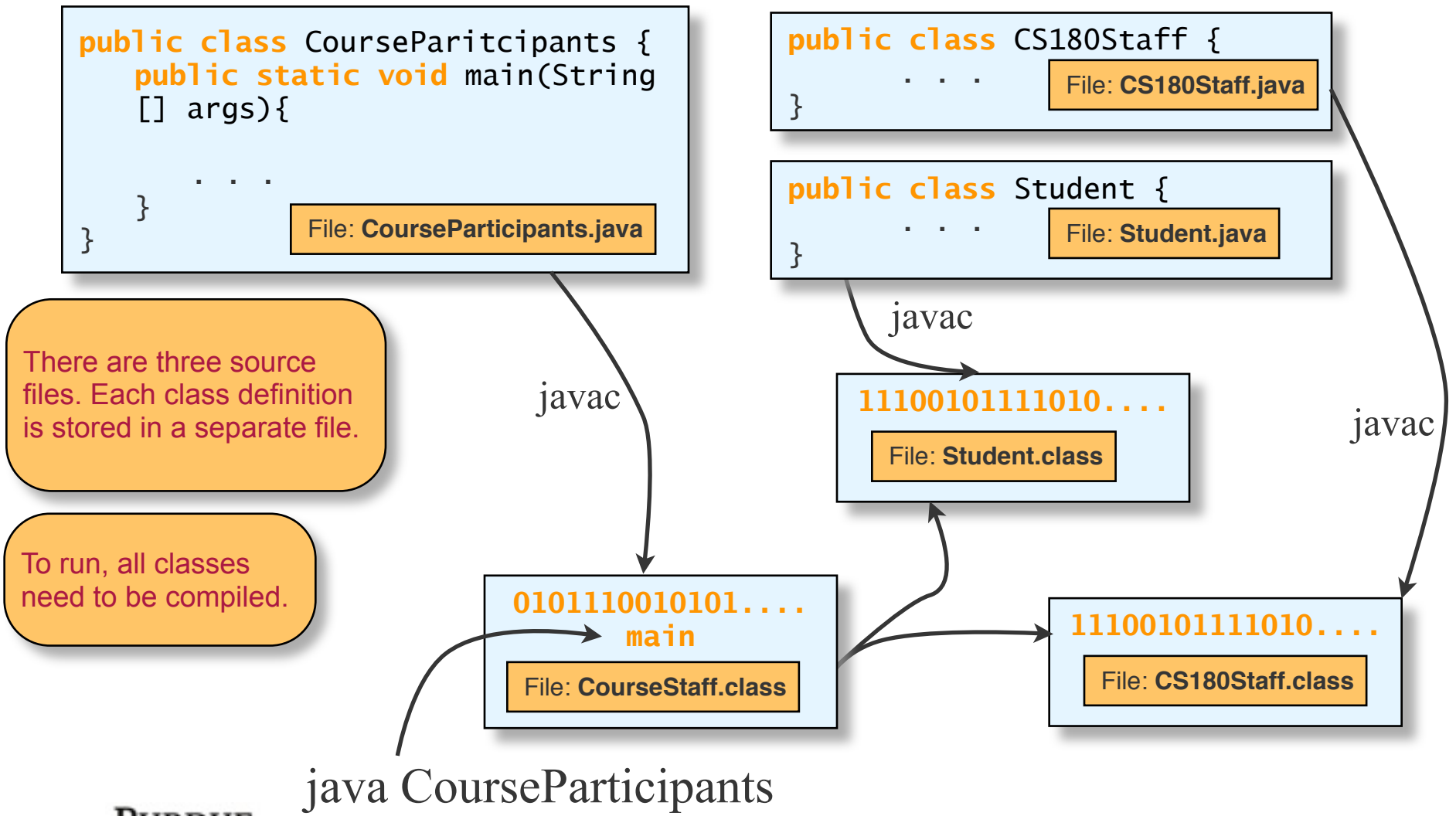
```
// ...
  public String getId(){
    return id;
  }

  public double getGpa(){
    return gpa;
  }

  public void setGpa(double g){
    gpa = g;
  }
}
```

Controller class: CourseParticipants

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

      CS180Staff instructor, coordinator,ta;
      Student student1, student2;

      instructor = new CS180Staff();
      instructor.getDetails();
      . . .
      student1 = new Student();
      student1.getDetails();

      student2 = new Student();
      student2.getDetails();

      . . .
      student1.printNeatly();
      student2.printNeatly();
   }
}
```

Note: same method names but different behavior for CS180Staff and Student objects.

Files and Classes



Two types of methods

- There are two types of methods:
  - object methods
  - class methods
- A class method is defined **with** the **static** keyword.
- An object method is defined **without** the **static** keyword.
- An object method must been called on an object of the class

Calling Class Methods

- Methods defined with the **static** keyword are called Class Methods.
- These methods are called using the class name.

```
class ComputeSine {
  public static void main() {
     double sine, angleRad = 3.4;
     String text;

     text = JOptionPane.showMessageDialog(null, "Enter
Angle in Radians");
     angleRad = Double.parseDouble(text);

     sine = Math.sin(angleRad);
  }
}
```

Class names

Calling Object Methods

- Methods defined without the **static** keyword are called Object Methods.
- These methods can only be called on an object of the given class.

```
class Test {
  public static void main() {
      Student s;
      String n;

      s = new Student();
      s.getId();
    ☹ n = getName();
     }
}
```

getName() must be called on a Student object.

From Outside the Class

need the class name to call a class method

```
class Test {

  SampleClass obj;

☹ classMethodA();

  SampleClass.classMethodA();

☹ objectMethodA();

☹ obj.objectMethodA();

  obj = new SampleClass();

  obj.objectMethodA();

}
```

```
class SampleClass {

  public static void classMethodA(){
    . . .
  }

  public void objectMethodA(){
    . . .
  }

}
```

obj does not reference a valid SampleClass object.

obj does not reference a valid SampleClass object.

From Within the Class

A **class** method can directly call another **class** method of the same class.

A class method needs an object of the class in order to call an object method

An object method needs to use the class name in order to call a class method

An **object** method can directly call another **object** method of the same class without specifying an object.

```
class SampleClass {
  public static void classMethodA(){
    SampleClass obj;

    classMethodB();
    objectMethodA();☹
    obj = new SampleClass();
    obj.objectMethodA();
  }

  public static void classMethodB(){
    . . .
  }

  public void objectMethodA(){
    classMethodA();☹
    SampleClass.classMethodA();
    objectMethodB();

  }

  public void objectMethodB(){
    . . .
  }
```

PURDUE UNIVERSITY

Calling Methods of Same Class

```
class Student {
  private String id;
  private String name;

  public void setName(String newName){
    . . .
  }
  private void setId(String newID){
    setName("ID changed");
    . . .
  }
  public String getId(){
    . . .
  }
```

No object specified.

setName() will use the same object as the one on which setId() was called.

Remember ...

- A class method needs a class name when called from
  - outside the class; or
  - a object method of the same class
- An object method needs to be called on an object of that class
  - when called from another object method of the same class, this object is implicit and not specified explicitly.
- Note: main is a class (static) method.

Initialization

```
class Student {
  private String name = "Unknown", id = "?";
  private double gpa= 0.0;
```

- Works for name and gpa, but is misleading. Can set it later.
- What about ID? We won't be able to change it. Thus, we need an id at object creation time.
- For this we use a special method called a *Constructor* -- this is called when an object is created using the **new** keyword.

A Constructor

```
public Student (  ) {
    . . .
}
```

- A constructor is a method defined within the class (like any other method).
  - it may or may not take any arguments
- However,

  it should always be a `public` method
  - it has no return type
  - its name is the same as the class name
- It is not necessary to define a constructor -- in this case the compiler creates a default constructor.

StudentV2 (with a Constructor)

```
class StudentV2 {
  private String name, id;
  private double gpa;

  public StudentV2(String studentID){
    id = studentID;
    name = "Unknown";
    gpa = 0.0;
  }

. . .
}
```

Using StudentV2 Objects

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

      StudentV2 student;

      student = new StudentV2("3478734");
      student.printNeatly();
   }
}
```

- Since the constructor for Student expects a String argument, we can no longer create student objects without providing this argument.
- Now a student object will have the given ID when created.

Constructors

- Each class must have a constructor if we are to create objects of the class.
- A constructor without arguments is called a default constructor.
- If a class does not define any constructors then a default constructor is automatically provided by the compiler.
- If any constructors are defined in the class, no default constructor will be provided.

Multiple Constructors

- What if we want to optionally allow objects to have an ID and a name at initialization?
- We can define two versions of constructors.

```
class StudentV3 {
  . . .
  public StudentV3(String studentID){
    id = studentID;
    name = "Unknown";
    gpa = 0.0;
  }

  public StudentV3(String studentID, String sName){
    id = studentID;
    name = sName;
    gpa = 0.0;
  }
. . .
```

Multiple Constructors

- Which one is executed to create a new object?
- Depends upon how many (and what types of) arguments are passed to it.
  - More later.

```
class Test {
    public static void main(String[] args){
    StudentV3 student;

 ☹  student = new StudentV3();

    student = new StudentV3("677632");

    student = new StudentV3("7658478", "John Doe");

  }
}
```

This version not available now!

Return values

- If the return type of a method is **void**, it returns nothing.

Otherwise, it must end with a **return** statement:

```
return <expression>;
```

- The expression may be an identifier, an expression, or a literal.
- The type of the expression must be compatible with the return type of the method.
- A **return** causes the method to return to the caller.

Problem

- *Extend the student class so that it recomputes the gpa when grades are reported. The class will not keep track of grades -- only the gpa.*
  - *recordGrade(); will be called to ask the Student object to record a grade. It should prompt for the number of credits for the course, and the grade expressed as an integer (4 for A, 3 for B, etc…) and then recompute the GPA.*

The StudentV4 Class

```
class StudentV4 {
 . . .
  private int totalCredits;
  public StudentV4(String studentID, String sName){
     . . .
    totalCredits = 0;
  }
 public void recordGrade(){
    int grade, credit;
    credit = Integer.parseInt(JOptionPane.showInputDialog(null,
                                "Enter Number of Credits");
    grade = Integer.parseInt(JOptionPane.showInputDialog(null,
                                "Enter Grade (as integer)");
    recomputeGpa(credit, grade);
  }

  public void recomputeGpa(int newCredit, int newGrade){
    double totalGradeCredits;

    totalGradeCredits = gpa * totalCredits;
    totalCredits += newCredit;
    totalGradeCredits += newCredit * newGrade;
    gpa = totalGradeCredits / totalCredits;
  } . . .
```

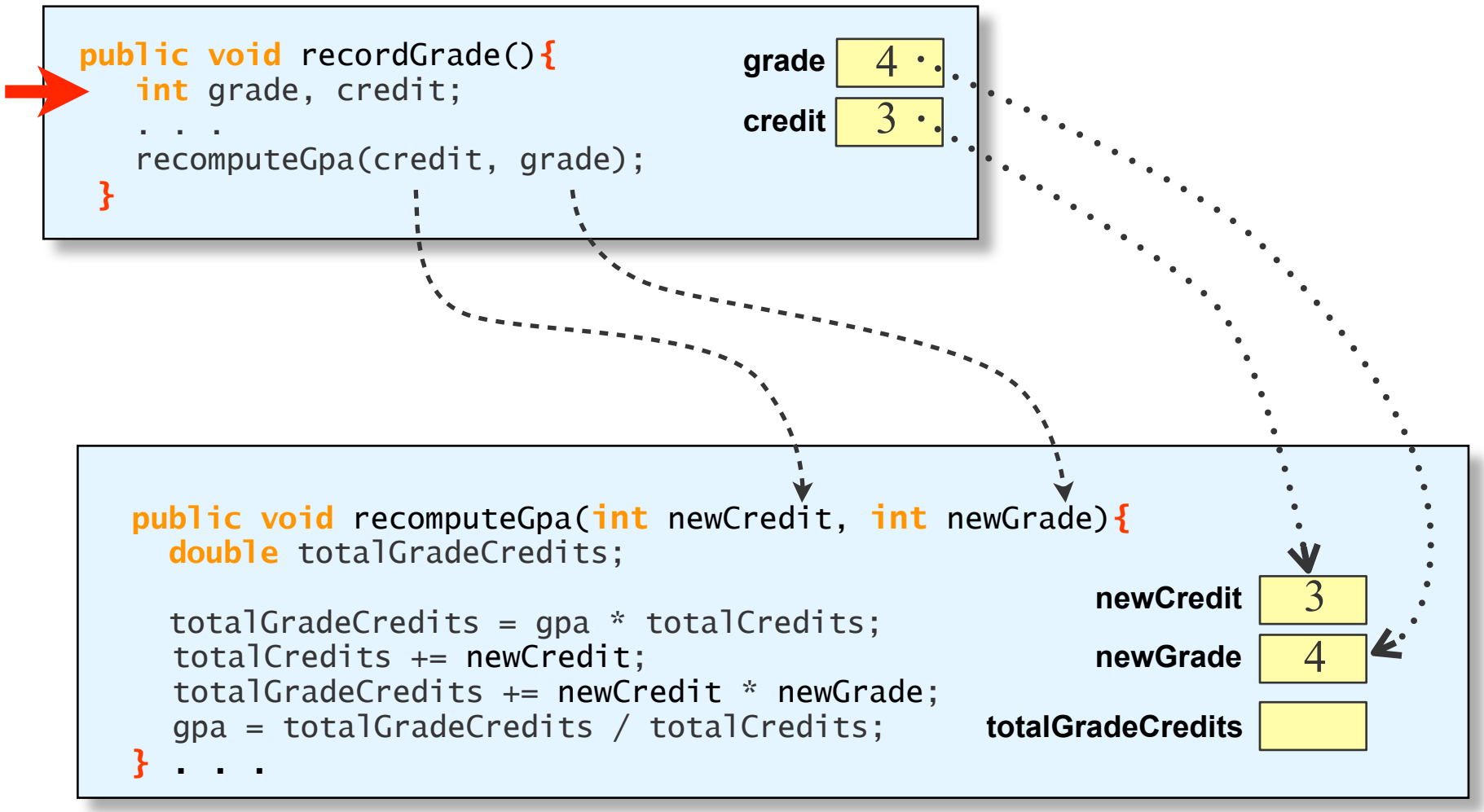
Using StudentV4

```
class TestStudentV4 {
   public static void main (String[] args) {
      StudentV4 jane;

      jane = new StudentV4("2342342", "Jane Doe");

      jane.recordGrade();
      jane.printNeatly();

      jane.recordGrade();
      jane.printNeatly();
   }
}
```

Method Arguments

- If a method takes arguments, every call to the method must provide values for these.
- The number and type of arguments expected by a method are declared by the method.
- Each time a method is called
  - New storage is created for its arguments
  - These arguments are initialized with copies of the values from the method call.
  - The program begins execution at the beginning of the method.

Initializing arguments



Matching arguments

```
public void methodA(){
    int i, j;
    double x, y;
    . . .
    methodB(i, x);
☹   methodB(x, i);
☹   methodB(i);
    methodC(i, x);
    methodC(x, i);
☹   methodD(i, x);
    methodD(i, x, 5);
}
```

```
public void methodB(int a, double b){
    . . .
}
```

```
public void methodC(double a, double b) {
    . . .
}
```

```
public void methodD(int a, double b, int c){
    . . .
}
```

Pass by value

```
public void methodA(){
    int i;
    i = 1;

    doubleUp(i);
    System.out.println("i = " + i);
}
```

i = 1

```
public void doubleUp(int a){
   a = 2 * a;
   System.out.println("a = " + a);
}
```

a = 2

- Only a copy of the value of i is passed. Changes to a do not affect the value of i.
- This is called *Pass by value*

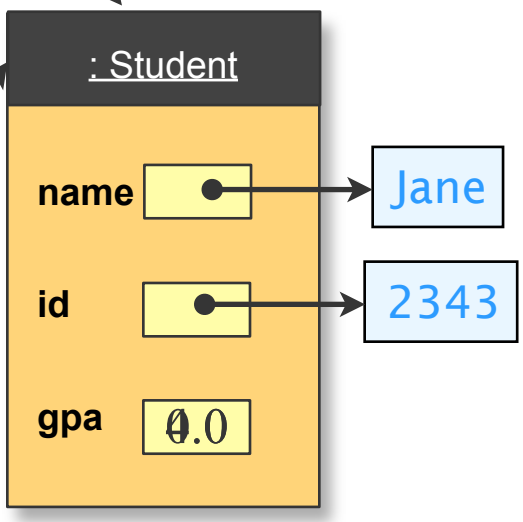
Objects as arguments

- Methods can take object arguments as well.
- The class of the object in the call must be compatible with the class of the object in the method declaration.
  - for now this means the same class
- As with assignment of reference types, only the reference is copied over.
  - the object being referenced is not copied!

Passing Objects

```
public void methodA(){
    Student student;

    student = new Student("2343", "Jane");
    changeGpa(student);
    System.out.println("Gpa = " + student.getGpa
());
```

```
Gpa = 4.0
```




```
public void changeGpa(Student s){
  s.setGpa(4.0);

}
```

Returning Objects

- As with arguments passed to a method,
  - when a method returns a primitive data type, the value is returned.
  - when a method returns a class data type, the reference to an object is returned.
- For example,
  - A constructor returns the reference to the newly created object.

Protecting Object Data

```
class Test {
  public static void main (String[] arg){
    Student s = new Student("4343");
    s.id = "234";

    s.name = "Jane";

    s.setID("234");

    s.setName("Jack");

    System.out.println(s.getName() +

    s.getID());
```

```
class Student {
  public String id;
  public String name;

  public void setId(String newId){
    . . .
  }

  public void setName(String newName){
    . . .
  }

  public String getId(){
    . . .
  }
  public String getName(){
    . . .
  }
}
```

ID can be changed from outside!

Encapsulation

- One of the key benefits of OOP
- Limit who can view/modify what data members and how
  - avoids accidental or intentional errors
- Improves program reliability and reuse
- Achieved by
  - hiding data members from outside the class
  - limiting which methods can be called directly from outside the class
  - using **public** and **private** modifiers

Visibility modifiers

- A data member or method that is declared **public** can be accessed by the code in any class.
- A **private** data member can only be accessed code that is part of the same class.
- A **private** method can only be called from code that is part of the same class.

Protecting Object Data

```
class Test {
  public static void main (String[] arg){
    Student s = new Student("4343");
  ☹ s.id = "234";

    s.name = "Jane";

  ☹ s.setID("234");

    s.setName("Jack");

    System.out.println(s.getName() +

    s.getID());
```

```
class Student {
  private String id;
  public String name;

  private void setId(String newId){
    . . .
  }

  public void setName(String newName){
    . . .
  }

  public String getId(){
    . . .
  }
  public String getName(){
    . . .
  }
}
```

id and setId() are inaccessible.

Guidelines

- Implementation details (data members) should be **private**
  - Use accessor/mutator methods
- Internal methods should be **private**
- Constructors are usually **public**
- Constants may be made **public** if useful (e.g. Math.PI)
- Default value is **public**.

Accessor and Mutator Methods

- Since most data members are usually defined to be private, it is common practice to provide methods to read and modify the values of data members.
  - Accessor methods are methods that retrieve the value of private data members. E.g., getName(), getId()
  - Mutator methods are methods that modify the value of private data members. E.g., setName ().

Identifier types

- Identifiers can be declared almost anywhere in a program.
- There are three main types of declarations:
  - Data members of a class
    - Declared outside any method
    - Usually at the beginning of the class definition
  - Formal parameters of a method
  - Local variables inside a method

Identifier extent and scope

- Each identifier refers to a piece of memory.
- That piece is reserved upon declaration.
- The lifetime of this reservation is called the extent of the identifier.
- The ability to access this location from a given line of code is called scope.
- Important to understand both.
- Extent and scope depend upon the type of variable and its declaration.

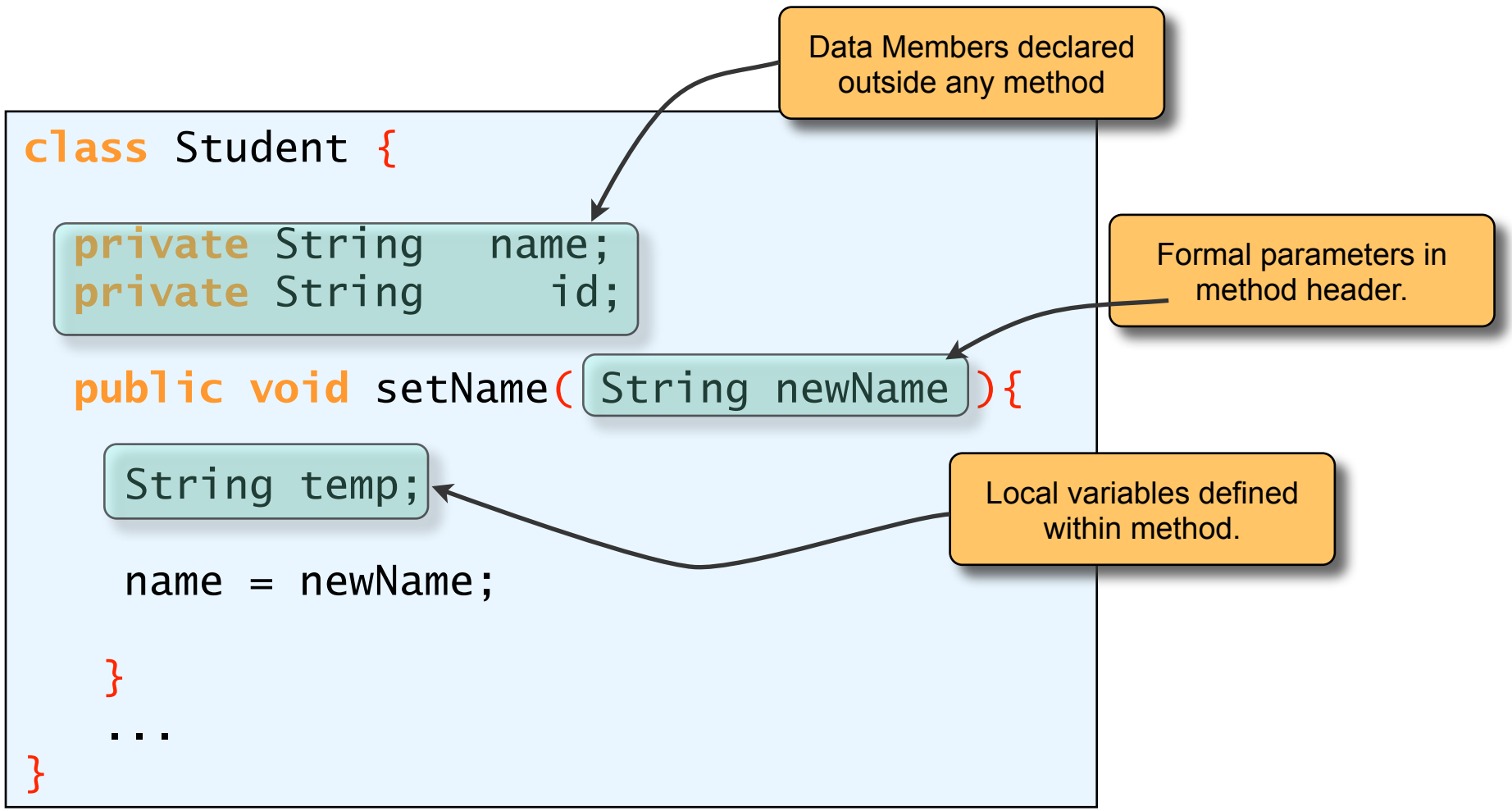
Extent

- Object data members
  - created when an object is created (by **new**)
  - destroyed when the object is garbage collected (no more references to it)
  - must be unique within each class
- Formal parameters
  - created each time the method is called
  - destroyed when the method finishes execution
  - must be unique for each method
- Local variables
  - created upon declaration
  - destroyed at end of block
  - must be unique for each block,
- Limiting extent allows compilers to reuse space

Which one do we mean?

- It is legal to reuse the name of a data member as a formal parameter, or a local variable.
- Each use of the identifier in a method is matched with exactly one of these as follows:
  - A local variable, or parameter, if it exists.
  - A data member, otherwise.
- Thus, a data member can be masked!
- Can lead to subtle errors.

Identifiers



Masked Data Member

```
class Student {

    private String     name;
    private String      id;

    public setName(String name) {

        String temp;

        name = name;

    }
    ...
}
```

Refer to formal parameter, not data member.

Masked Data Member 2

```
class Student {

    private String     name;
    private String      id;

    public setName(String newName) {

        String name;

        name = newName;

    }
    ...
}
```

Refers to local variable, not data member.

Remember, ….

- A local variable can be declared just about anywhere!
- Its scope (the area of code from where it is visible) is limited to the enclosing braces.
- Statements within a pair of braces are called a block.
- Local variables are destroyed when the block finishes execution.
- Data members of a class are declared outside any method. Their scope is determined by public and private modifiers.