UNIVERSITY of WISCONSIN
LA CROSSE
COMPUTER SCIENCE

Week 01:
Introduction

CS 220: Software Design II — D. Mathias

What is CS 220?

Second semester programming course

Covers…

- leftover topics from 120 (e.g., abstract classes/interfaces, file i/o, 2D arrays)
- data structures (e.g., lists, stacks, queues, sets, maps)
- algorithms and their analysis (e.g., searching, sorting)

A Typical Week

Monday	Tuesday	Wednesday	Thursday	Friday
lecture	lecture	lab (Wing 016)		in-class ex.

We will meet in lab on Wednesdays

Labs

Most weeks in Wing 016

Computers available there

- will need your NetID/password (i.e., what you use to log into WINGS, email)
- or you can use your laptop

Labs

Graded in one of two ways:

1. show up and are on task the entire time: full credit

   no need to submit anything

   not on task the whole time: half credit

   texting or playing <insert name of relevant video game here>: no credit

2. **if you can't make it to lab**, submit the lab program(s) via email

   this option applies to sickness/isolation, required travel, etc

No labs will be dropped

Programming Assignments

Released (roughly) every third week - 5 assignments in total

Corresponds to the topic of the previous week(s), due in ~2 weeks

Larger programs than 120

When should you start working on an assignment?

Course Materials

On my website: https://cs.uwlax.edu/~dmathias/cs220.html

I don’t post materials on Canvas

On Canvas, you will find:

- Announcements (check daily)
- Assignment due dates
- Assignment grades

Syllabus

On my website - **read it on your own**

- covers many things
- bring questions tomorrow
    - seriously, I expect you to read it and ask questions

Office Hours

Virtual only

- Monday 11:00 - 12:00
- Wednesday 2:15 - 3:15
- Friday 11:00 - 12:00

Zoom link is on the syllabus and multiple places on my website

Office Hours

Opportunity to…

- clarify material from class
- clarify requirements of assignments
- work on problem solving for programs
- get debugging help - but you need to become proficient at debugging

Not an opportunity for me to write your code

I will answer questions, and then ask you to grapple for a little bit with the program using new information/understanding of the material

- So start assignments early!

Expectations

Comfortable with 120 material

Competent with problem solving techniques

Self-sufficient in generating examples for studying/programming

- questions in office hours will work best if you bring these along

Format code correctly

Staying Afloat

Expect to spend ~12 hrs/week

Work consistently, a little every day

- **start assignments early**
- work through additional exercises

Class builds on itself, so solidify earlier concepts

Start assignments early

Attend office hours when needed

Start assignments early

Make friends in the class and form study groups

Start assignments early

Productive Work

Start assignments early

- fixing code constitutes ~50% of time spent on a project[1]
- ~60% of defects exist when understanding/conceptualizing the problem statement[1]

Spend time thinking about the problem, sketching out solutions in English

- helps clarify your understanding
- happy to discuss your reasoning

This process requires you to start assignments early

1: http://programmers.stackexchange.com/questions/91758/debugging-facts-and-statistics

Office Hours

Opportunity to…

- clarify material from class
- clarify requirements of assignments
- work on problem solving for programs
- get debugging help

Not an opportunity for me to write your code

I will answer questions, and then ask you to grapple for a little bit with the program using new information/understanding of the material

- start assignments early!

A Note on Working Together…

Working together is **encouraged**

- develop understanding of the problem
- swap ideas
- correct technical understanding of code constructs

Do **not** share code

Do **not** look at someone's code

Do **not** copy and paste someone's code (even if it's just a few lines/a method)

Do **not** write code together

When In Doubt

Write code individually

The person with working code should be looking at the problematic code

Talk to me

Start assignments early!

Introductions

Groups of 3-5

Introduce yourselves to one another

- name
- year in school
- major/minor
- do you start assignments early?
- what do you do when you're procrastinating?

Come up with one question you have for me

- about the course/computer science/me (that I would be willing to answer...)

UNIVERSITY of WISCONSIN
LA CROSSE
COMPUTER SCIENCE

Week 01:
Object-oriented Paradigm and Java Style

CS 220: Software Design II — D. Mathias

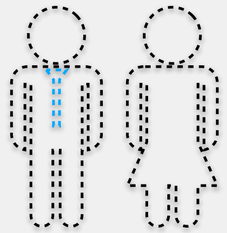
Object-oriented programs are comprised of **objects** from multiple classes **interacting**, mimicking how the **real world works**.

Classes

- Allow us to group together pieces of data that define a real world concept
  - even if they are of different datatypes!
  - e.g., a professor is made up of a first/last name, courses they teach…
- A *class* provides a definition of what pieces of data define a real world concept
- An *object* defines a particular *instance* of that class, providing concrete values

UWL as Object-Oriented Data

class



Professor
(*name*, *list of classes*, *office*)

objects

Allie Sauppé
CS120, CS364
Wing 214

David Mathias
CS120, CS224
Wing 212

Elliot Forbes
CS272, CS370
Wing 219

Jason Sauppe
CS225, CS371
Wing 207

Sam Foley
CS270, CS441
Wing 220

Tom Gendreau
CS340, CS442
Wing 211

Components of Classes

- Identifier
  - name of the class
  - should be singular, start with a capital letter (e.g., Professor, Student)
- Attributes
  - data that defines every object of that class type
- Methods
  - define the actions that can be taken with objects of that class type

Components of Classes

```
public class Professor {

    private String firstName;
    private String lastName;
    private String dept;
    private Course[] courses;

    public Professor(String fn, String ln) {
      this.firstName = fn;
      this.lastName = ln;
    }

    public String getDept() {
      return dept;
    }

    public void setDept(String dept) {
      this.dept = dept;
    }

}
```

only part of the class
(missing many details)

Components of Classes: Identifier

```
public class Professor {

    private String firstName;
    private String lastName;
    private String dept;
    private Course[] courses;

    public Professor(String fn, String ln) {
      this.firstName = fn;
      this.lastName = ln;
    }

    public String getDept() {
      return dept;
    }

    public void setDept(String dept) {
      this.dept = dept;
    }

}
```

Name of the class

Should be singular

Should start with a capital letter (e.g., Professor, Student)

Components of Classes: Attributes

```
public class Professor {

    private String firstName;
    private String lastName;
    private String dept;
    private Course[] courses;

    public Professor(String fn, String ln) {
      this.firstName = fn;
      this.lastName = ln;
    }

    public String getDept() {
      return dept;
    }

    public void setDept(String dept) {
      this.dept = dept;
    }

}
```

- Data that defines every object of that class type
- Variable declarations at a minimum
  - can also initialize/instantiate if needed
- Also referred to as *global variables*
  - have scope throughout the class
  - should always provide a visibility

Components of Classes: Methods

```
public class Professor {

    private String firstName;
    private String lastName;
    private String dept;
    private Course[] courses;

    public Professor(String fn, String ln) {
      this.firstName = fn;
      this.lastName = ln;
    }

    public String getDept() {
      return dept;
    }

    public void setDept(String dept) {
      this.dept = dept;
    }

}
```

Define the actions that can be taken with objects of that class type

Components of Classes: Constructor Method

```
public class Professor {

    private String firstName;
    private String lastName;
    private String dept;
    private Course[] courses;

    public Professor(String fn, String ln) {
      this.firstName = fn;
      this.lastName = ln;
    }

    public String getDept() {
      return dept;
    }

    public void setDept(String dept) {
      this.dept = dept;
    }

}
```

Method to create (*instantiate*) an object of this class type

Named the same as the class

Lacks a return type

Visibility

- Used to control access to classes, methods, and attributes
- Three options
  - public: can be accessed from any class
  - private: can only be accessed from its own class
  - protected: accessible to this class and child classes
- Visibility applies to classes, methods, and attributes
  - `public class Professor`
  - `public static void printArray(char[] arr)`
  - `private String firstName`

Visibility Rules of Thumb

- Classes are usually public
  - tend to only be useful to us if they can be accessed from other classes
- Attributes are usually private
  - don’t want people to change them at will
  - forces change through methods, which provide guarantees
- Methods are most likely public, but private is also common
  - public methods used to work with objects of that type
  - private methods used to help internal class functionality

Getter and Setter Methods

- Since attributes are usually private, need some way to access them
- *Getter methods* get the value of an attribute
- *Setter methods* set the value of an attribute
  - can be used to ensure the attribute is only set to sensible values
  - e.g., only possible values for birth month are 1-12
- Example for firstName attribute
  - public String getFirstName()
  - public void setFirstName(String fn)

Static vs Non-Static Methods

The *static* keyword controls whether a resource (e.g., method, variable) belongs to the *class* or an *object* of that class type

- static: do not need to have instantiated an object of that class type to use it
- non-static: must have an object instantiated of that class type

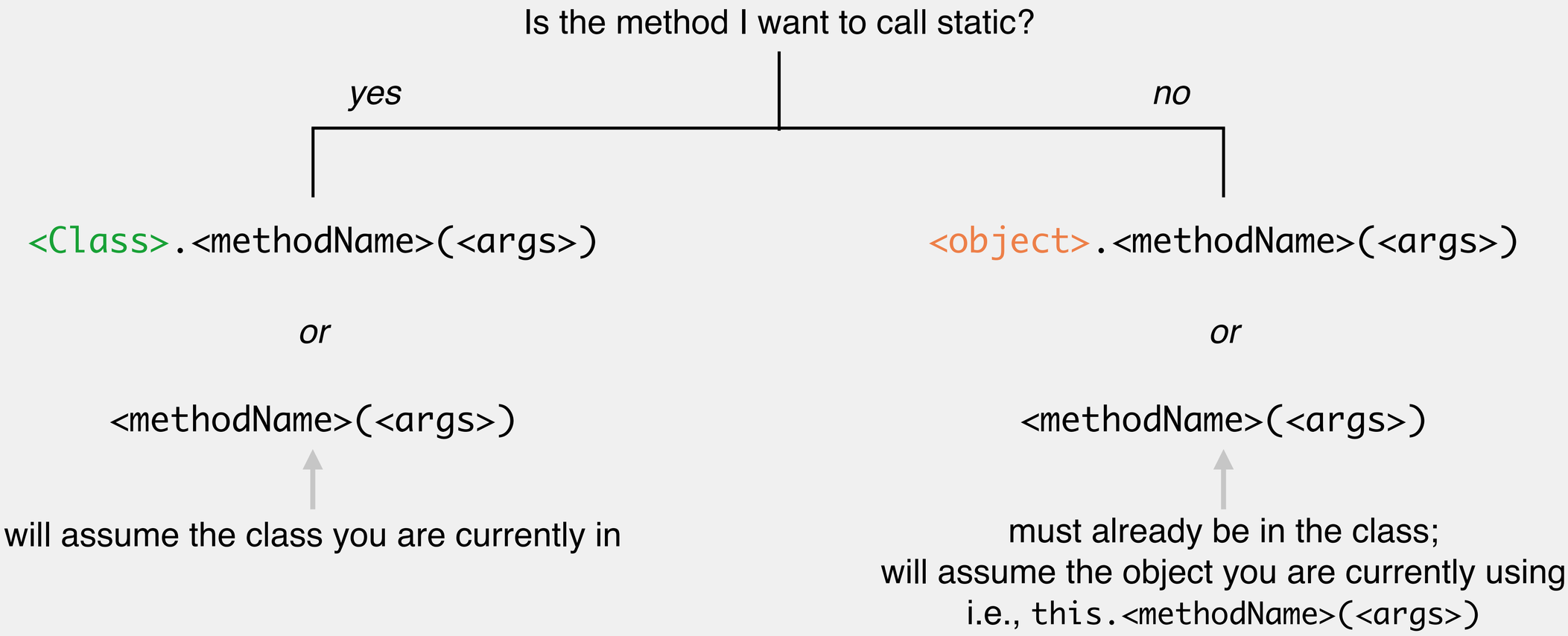
Overarching question: Do I need to know one or more attribute values from an object to use this?

- yes? non-static
- no? static

Static Rules of Thumb

- Generally, methods/variables will be non-static
  - conforms to object-oriented principles
- Static methods can only access static attributes
  - non-static methods can access all attributes
- Examples of static methods from Java:
  - everything from the Math class
  - `Math.pow(double x, int y)`
  - `Math.max(double x, double y)`

How to Call Methods



Steps to Creating a New Class

1. Class name
2. Attributes

   name, type, visibility, initialization/instantiation?
3. Constructor method

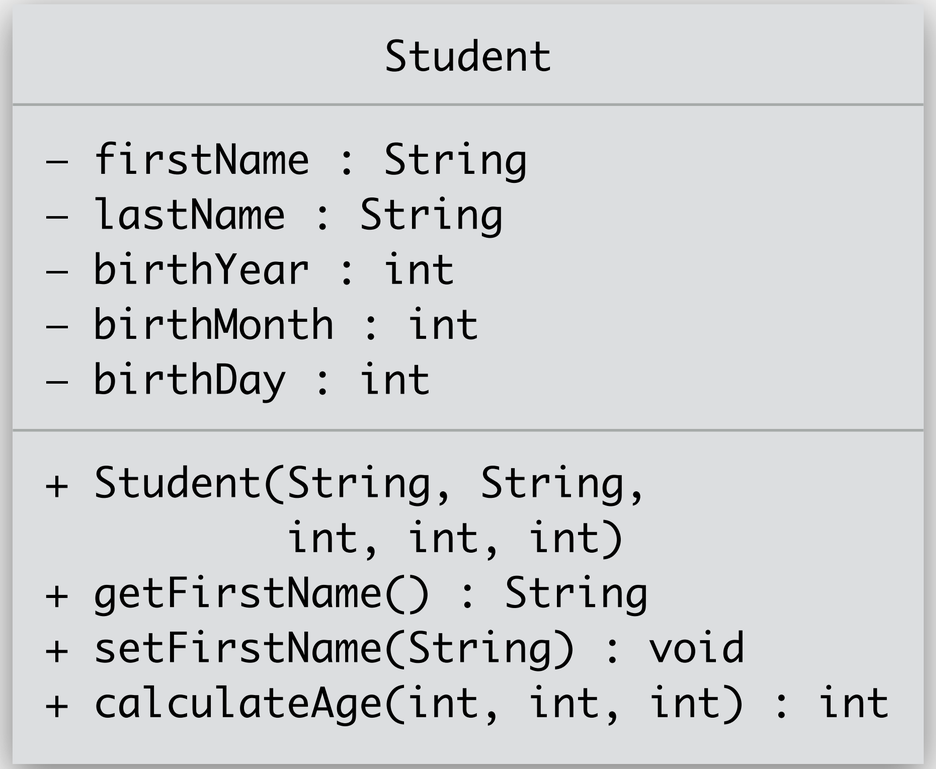
   parameters come from attributes
4. Other methods

   getters/setters, methods specified in requirements

Example: Creating a New Class

Write a new class called Student. Each object of this type will represent a single student at UWL. Students are defined by first, middle, and last name, a username (lastname.firstname), birthday, and a home address. Write the getter/setter methods for the first name and last name. Additional methods should also return their email address (username@uwlax.edu) and their age.

Class Diagram



Easy way to represent basic components of a class (name, attributes, methods)

Part of *unified modeling language* (*UML*)

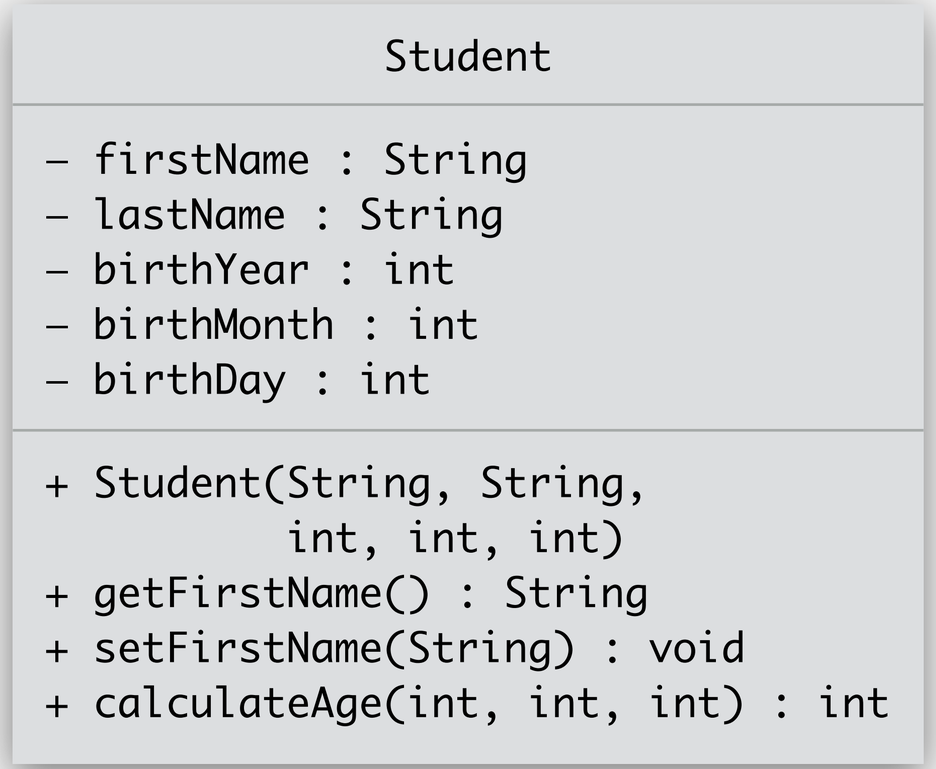
- used to communicate structure of programs

Visibility prefaces identifier

- + for public
- — for private
- # for protected

Static attributes/methods are underlined

Class Diagram



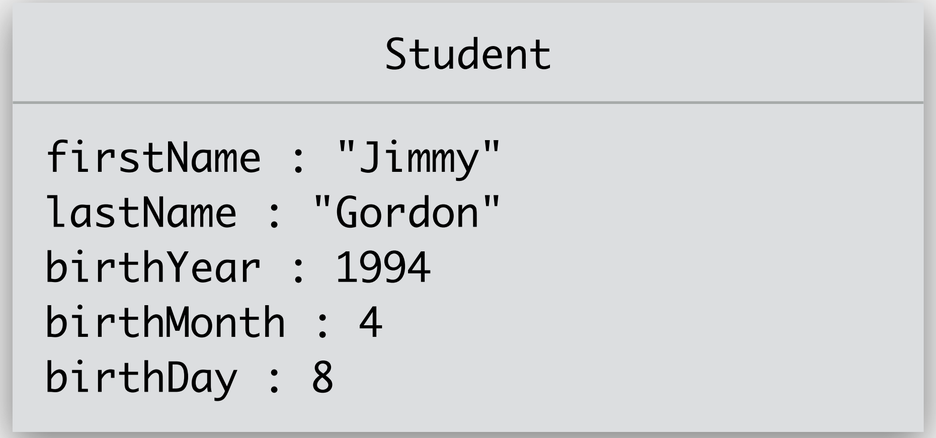
Attributes list type after colon

Methods list only parameter types

Return type appears after method, prefaced with a colon

- constructor will not list a return type
- list void if no return type

Object Diagram



Used to identify current state of object

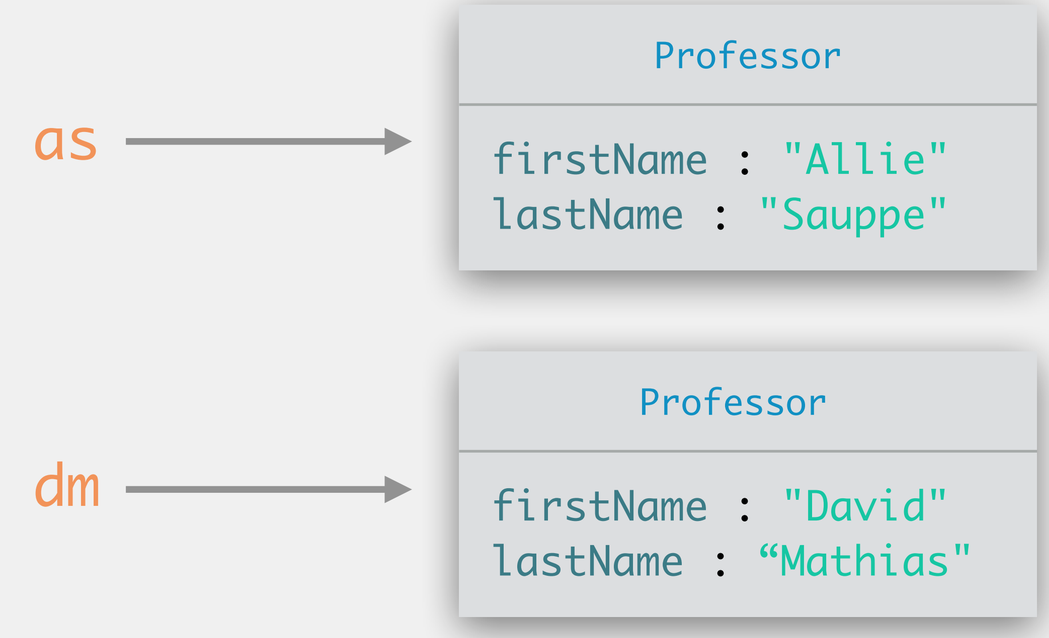
Lists current values for each attribute

Does not list methods

- do not change depending on object

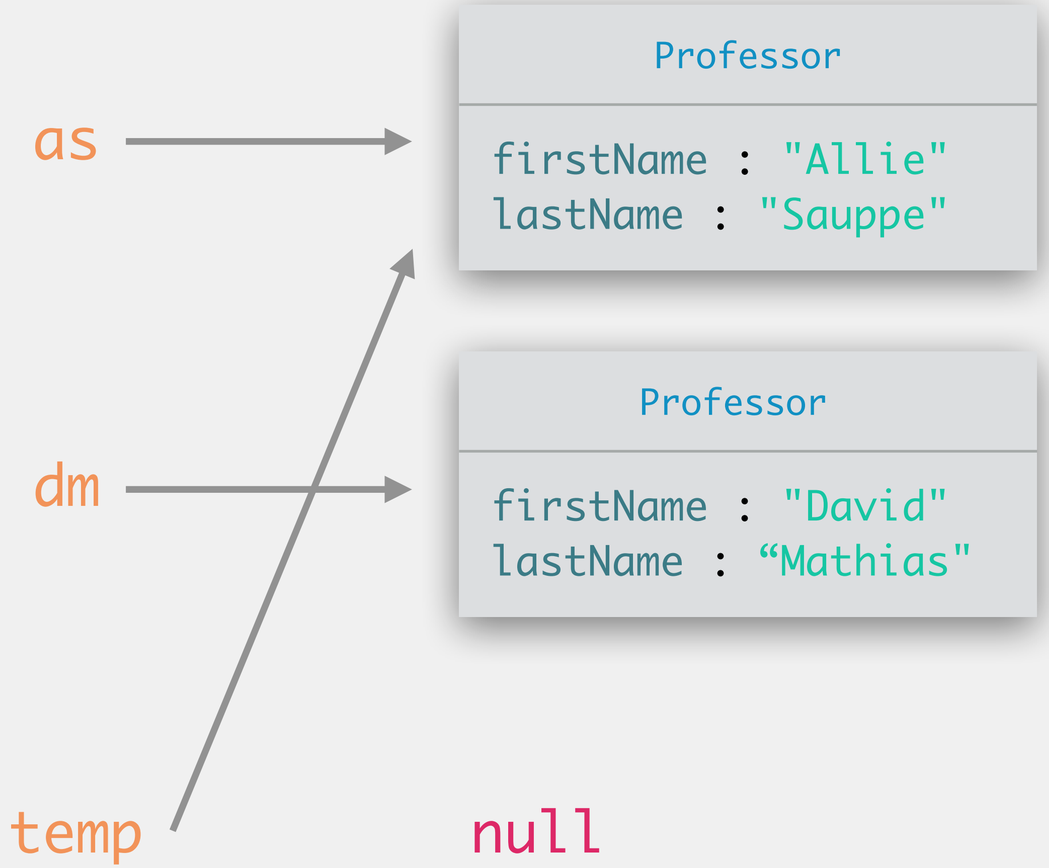
Object Tracing

```
>Professor as = new Professor("Sauppe", "Allie");

>Professor dm = new Professor("Mathias", "David");

>Professor temp;

>temp = as;
 as = dm;
 dm = temp;
 temp = null;
 System.out.println(as.getFirstName() +
      " " + dm.getFirstName());
```



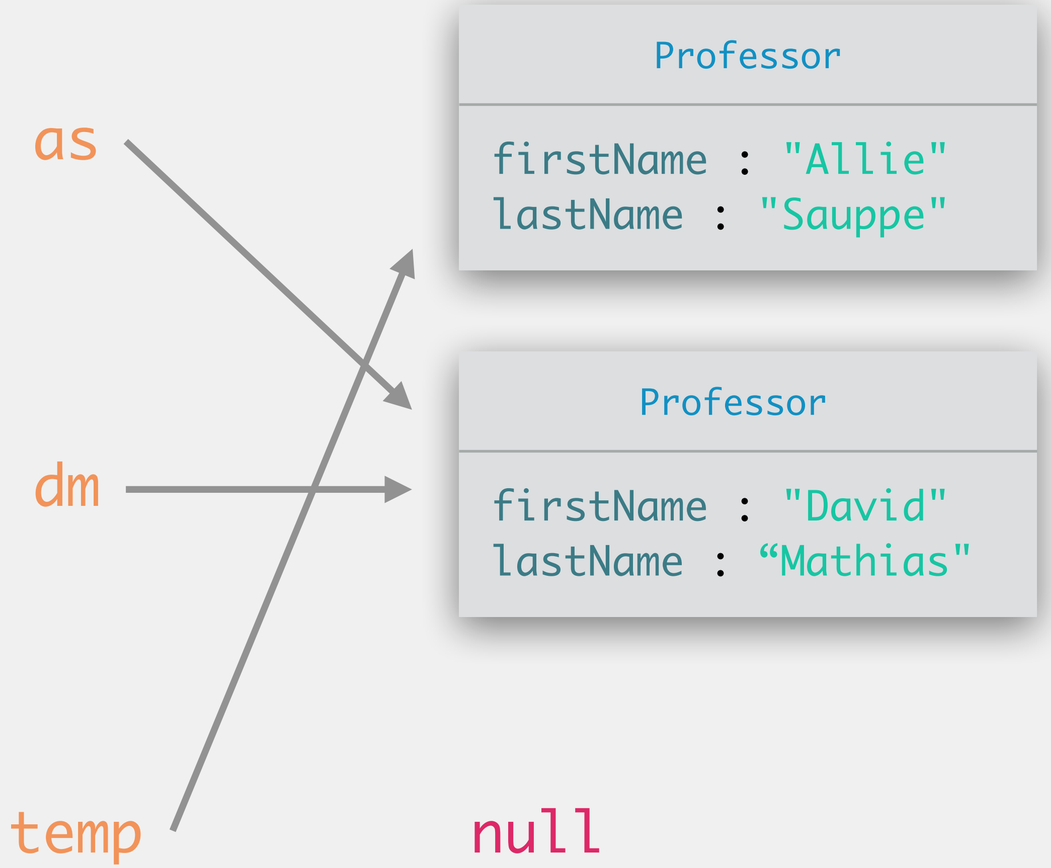
Object Tracing

```
Professor as = new Professor("Sauppe", "Allie");

Professor dm = new Professor("Mathias", "David");

Professor temp;

temp = as;
>as = dm;
dm = temp;
temp = null;
System.out.println(as.getFirstName() +
    " " + dm.getFirstName());
```



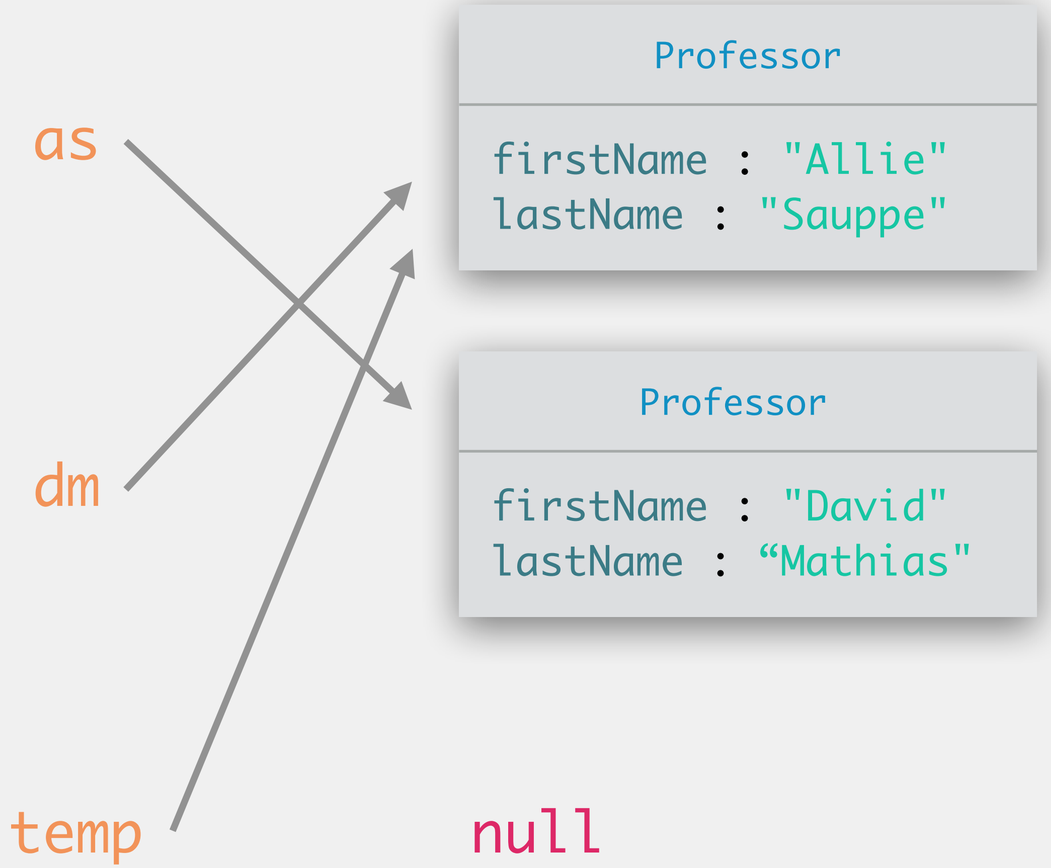
Object Tracing

```
Professor as = new Professor("Sauppe", "Allie");

Professor dm = new Professor(“Mathias", "David");

Professor temp;

temp = as;
as = dm;
>dm = temp;
temp = null;
System.out.println(as.getFirstName() +
     " " + dm.getFirstName());
```



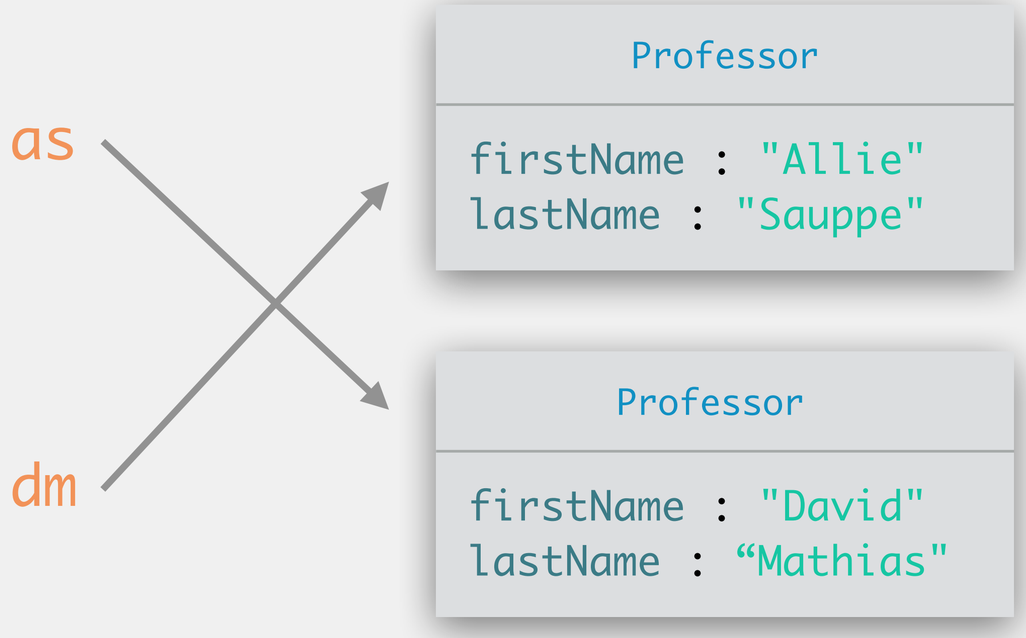
Object Tracing

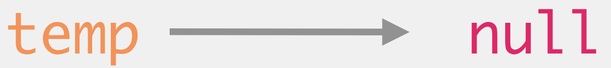
```
Professor as = new Professor("Sauppe", "Allie");

Professor dm = new Professor("Mathias", “David");

Professor temp;

temp = as;
as = dm;
dm = temp;
>temp = null;
System.out.println(as.getFirstName() +
     " " + dm.getFirstName());
```



Object Tracing

```
  Professor as = new Professor("Sauppe", "Allie");

  Professor dm = new Professor("Mathias", “David");

  Professor temp;

  temp = as;
  as = dm;
  dm = temp;
  temp = null;
> System.out.println(as.getFirstName() +
      " " + dm.getFirstName());
```

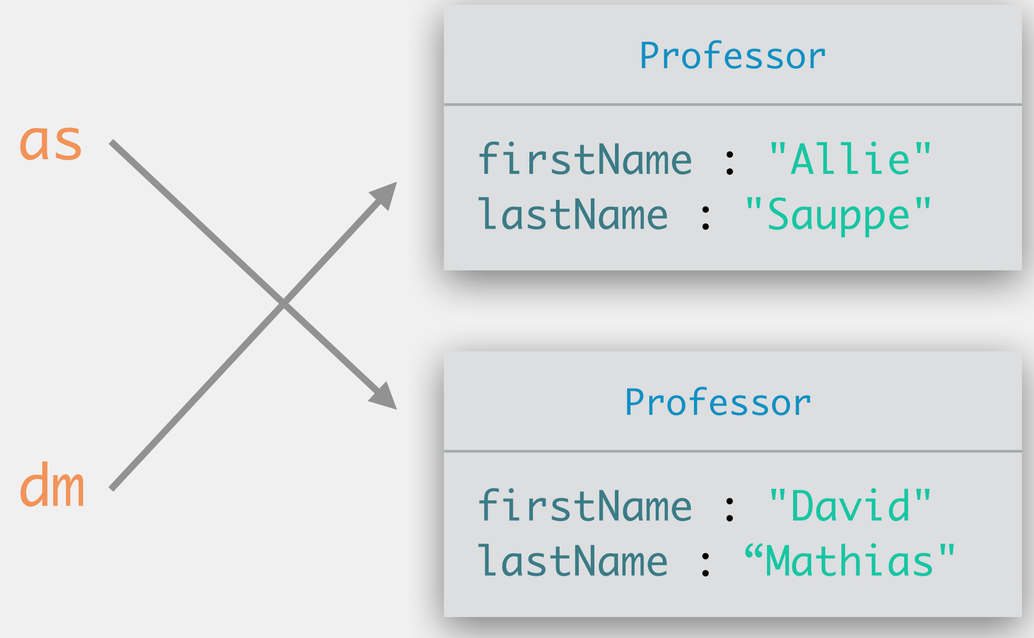


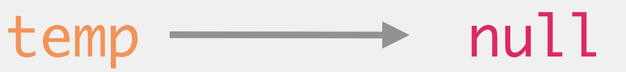


Object Tracing

```
Professor as = new Professor("Sauppe", "Allie");

Professor dm = new Professor("Mathias", “David");

Professor temp;

temp = as;
as = dm;
dm = temp;
temp = null;
System.out.println(as.getFirstName() +
    " " + dm.getFirstName());
```

```
David Allie
```





always treat variables of a class type and the objects they refer to as two separate entities

Object Tracing With Methods

```
Professor as = new Professor("Sauppe", "Allie");

Professor dm = new Professor("Mathias", “David");

as.renameProf("Allison");

dm.renameProf(“Dude");
```

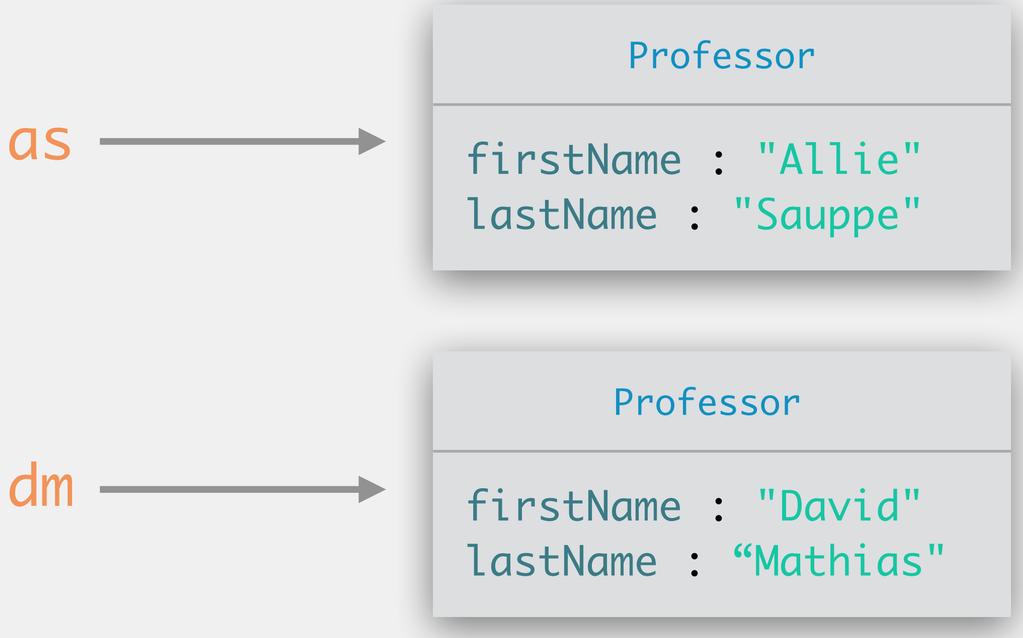
```
public void renameProf(String newName) {
    this.firstName = newName;
}
```

method contained
in the Professor class

Object Tracing With Methods

```
>Professor as = new Professor("Sauppe", "Allie");

>Professor dm = new Professor("Mathias", "David");

>as.renameProf("Allison");

 dm.renameProf(“Dude");
```

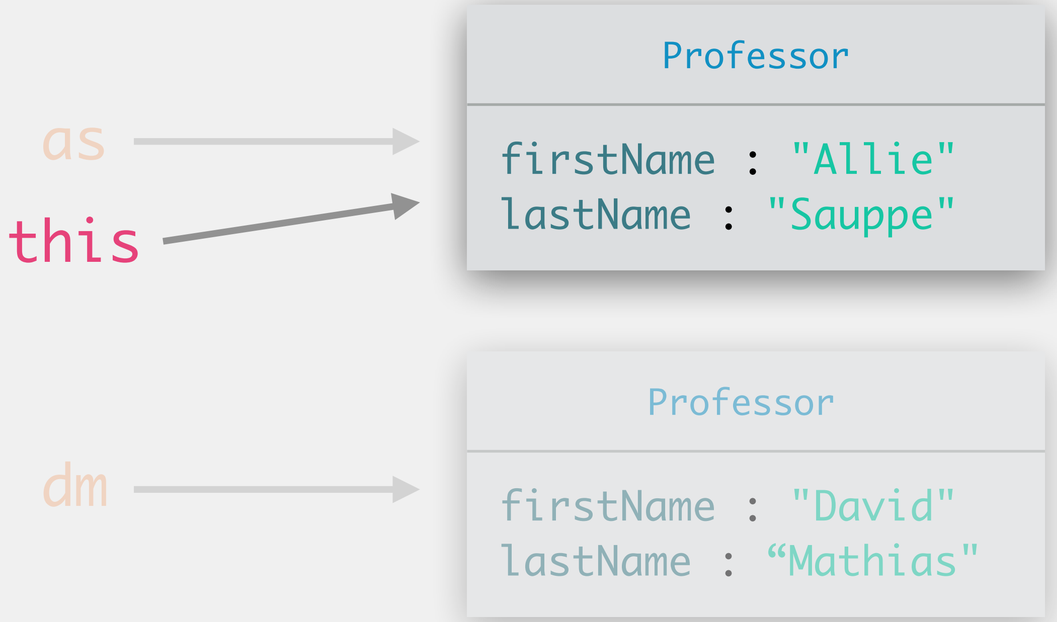
```
public void renameProf(String newName) {
    this.firstName = newName;
}
```



Object Tracing With Methods

```
  Professor as = new Professor("Sauppe", "Allie");

  Professor dm = new Professor("Mathias", “David");

>as.renameProf("Allison");

  dm.renameProf(“Dude");
```

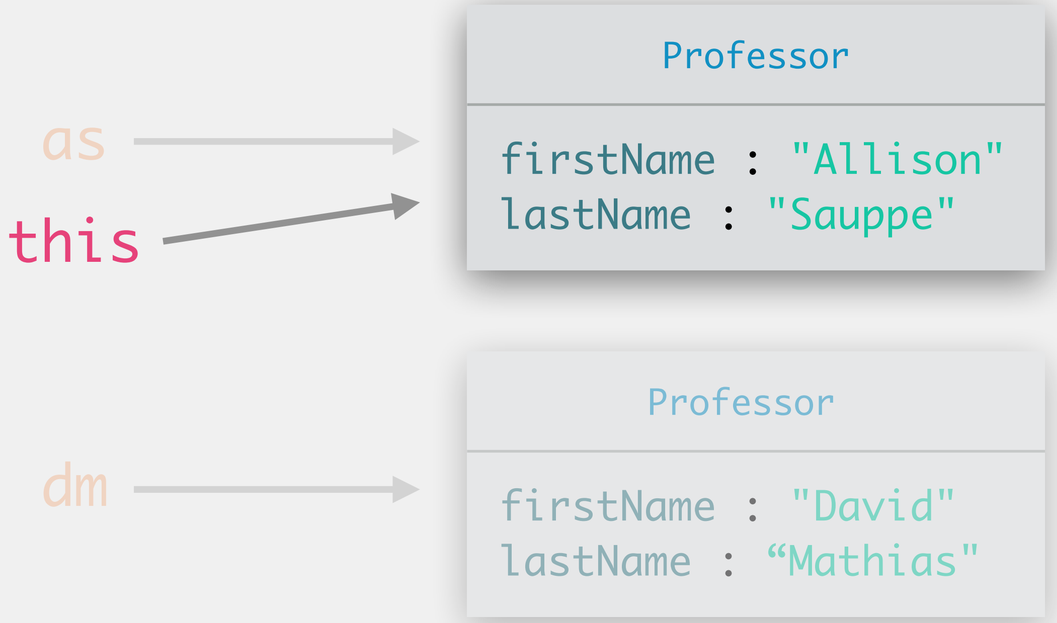
```
public void renameProf(String newName) {
   >this.firstName = newName;
}
```



Object Tracing With Methods

```
  Professor as = new Professor("Sauppe", "Allie");

  Professor dm = new Professor("Mathias", “David");

>as.renameProf("Allison");

  dm.renameProf(“Dude");
```

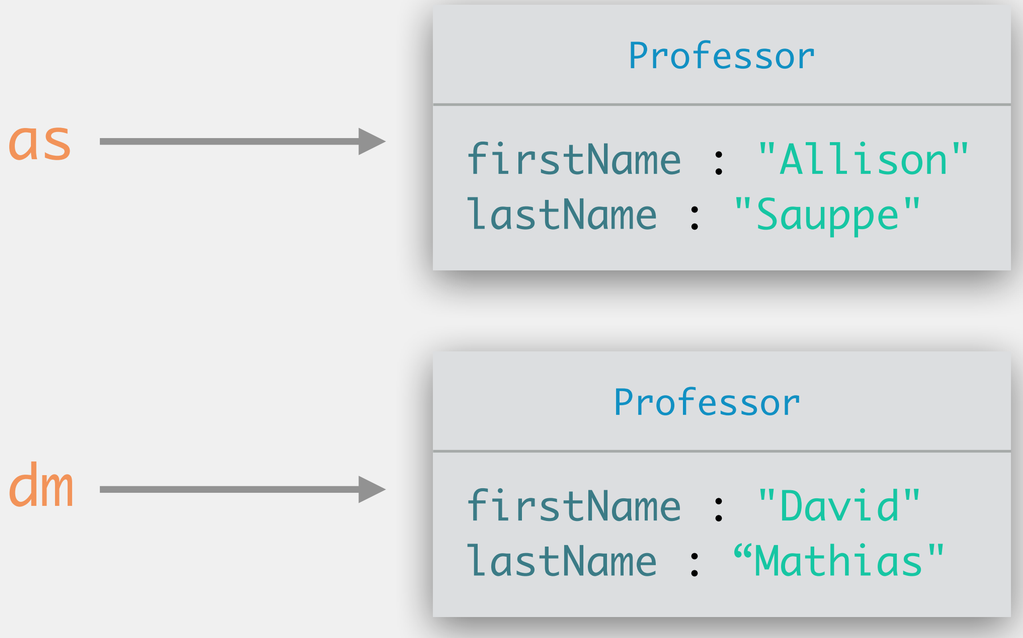
```
public void renameProf(String newName) {
    this.firstName = newName;
} >
```



Object Tracing With Methods

```
  Professor as = new Professor("Sauppe", "Allie");

  Professor dm = new Professor("Mathias", “David");

  as.renameProf("Allison");

>dm.renameProf(“Dude");
```

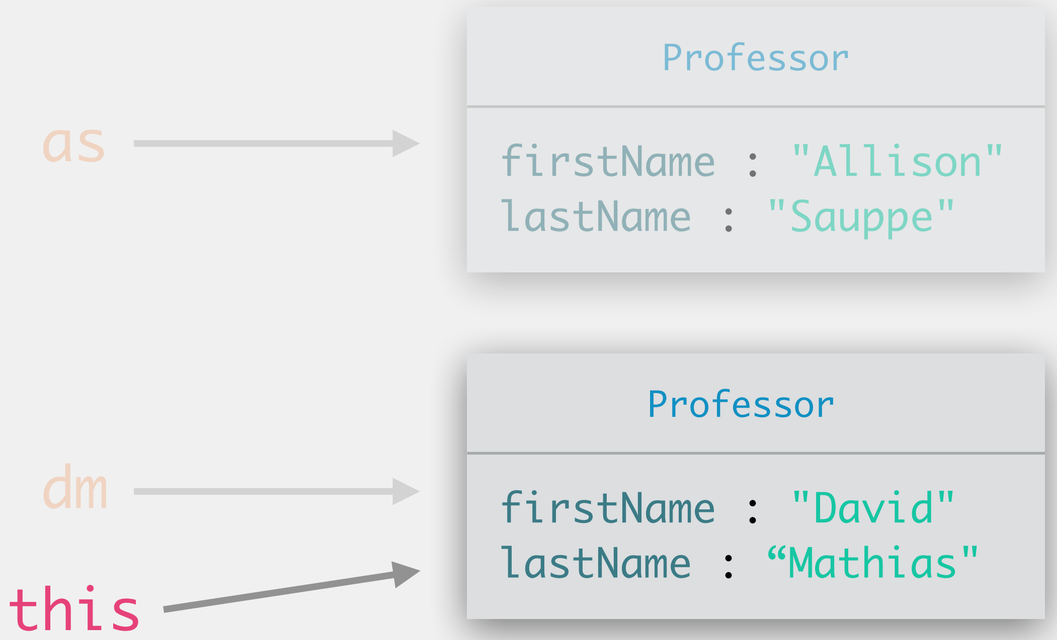
```
public void renameProf(String newName) {
    this.firstName = newName;
}
```



Object Tracing With Methods

```
  Professor as = new Professor("Sauppe", "Allie");

  Professor dm = new Professor("Mathias", "David");

  as.renameProf("Allison");

> dm.renameProf(“Dude");
```

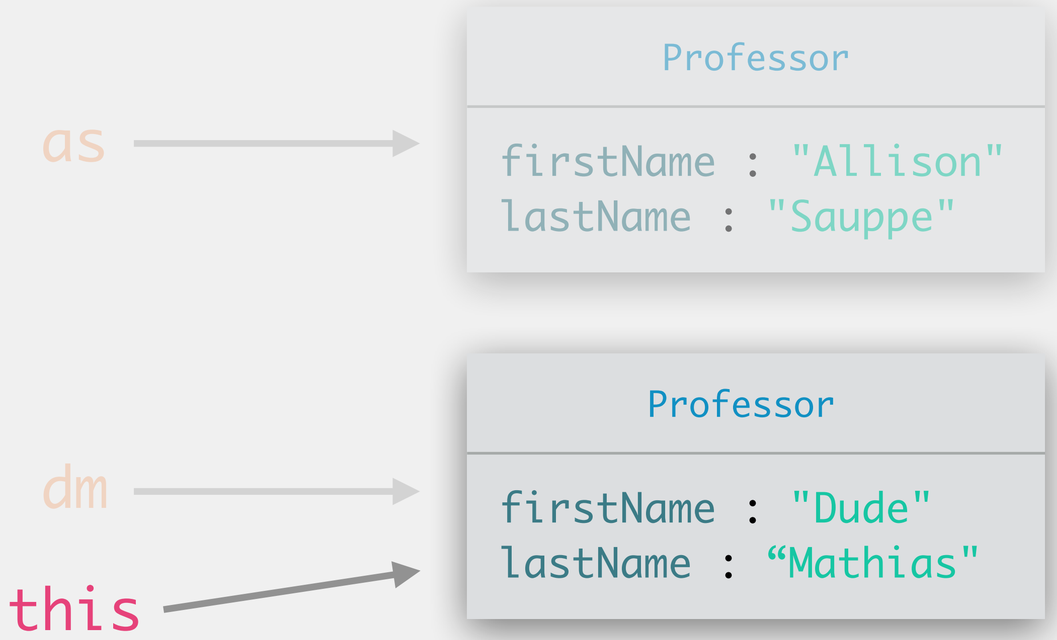
```
public void renameProf(String newName) {
   > this.firstName = newName;
}
```



Object Tracing With Methods

```
Professor as = new Professor("Sauppe", "Allie");

Professor dm = new Professor("Mathias", “David");

as.renameProf("Allison");

> dm.renameProf(“Dude");
```

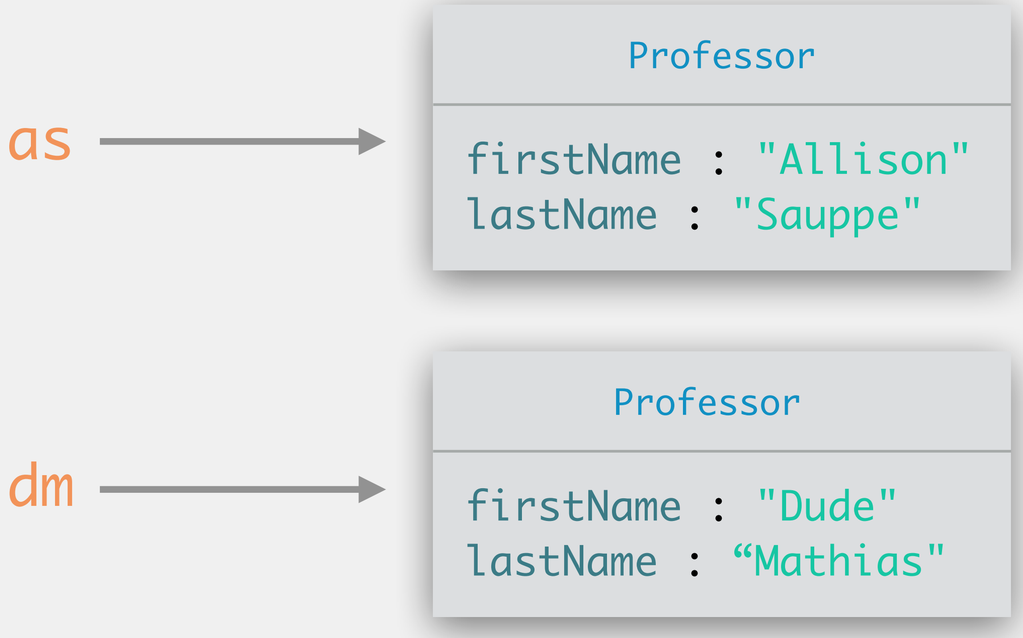
```
public void renameProf(String newName) {
    this.firstName = newName;
} >
```



Object Tracing With Methods

```
  Professor as = new Professor("Sauppe", "Allie");

  Professor dm = new Professor("Mathias", “David");

  as.renameProf("Allison");

  dm.renameProf(“Dude");
>
```

```
public void renameProf(String newName) {
    this.firstName = newName;
}
```



programs are comprised of **classes**

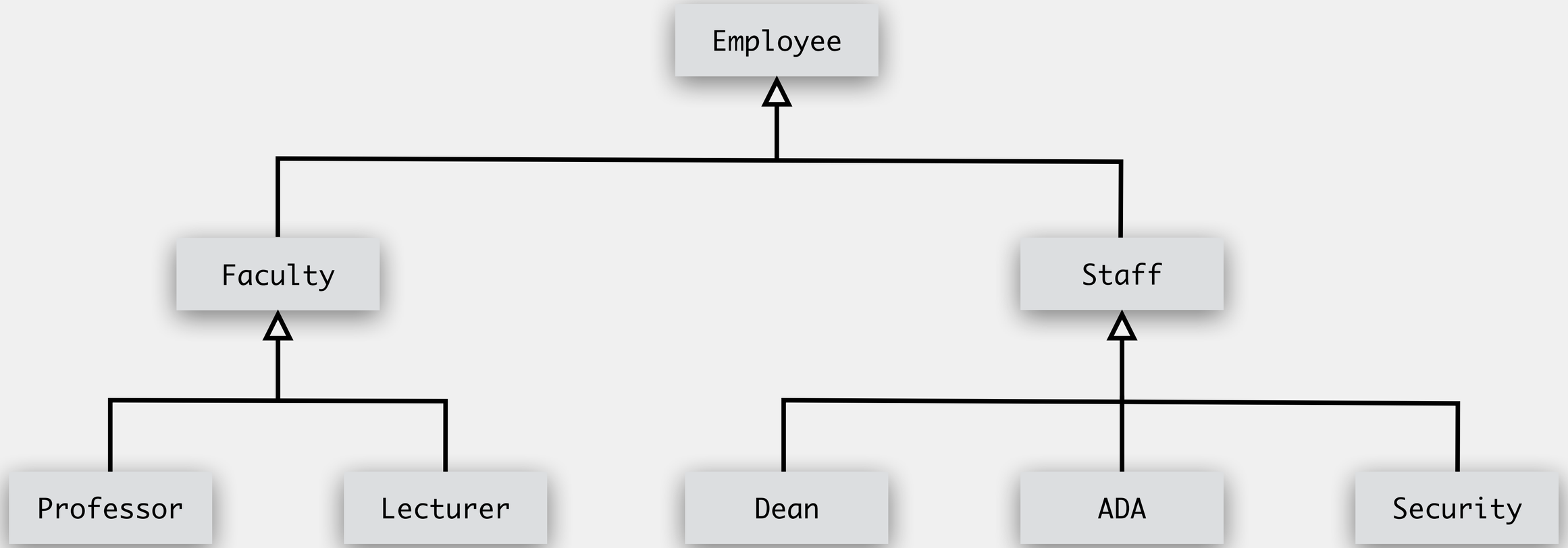
classes are comprised of **attributes + methods**

methods are comprised of **basic code**

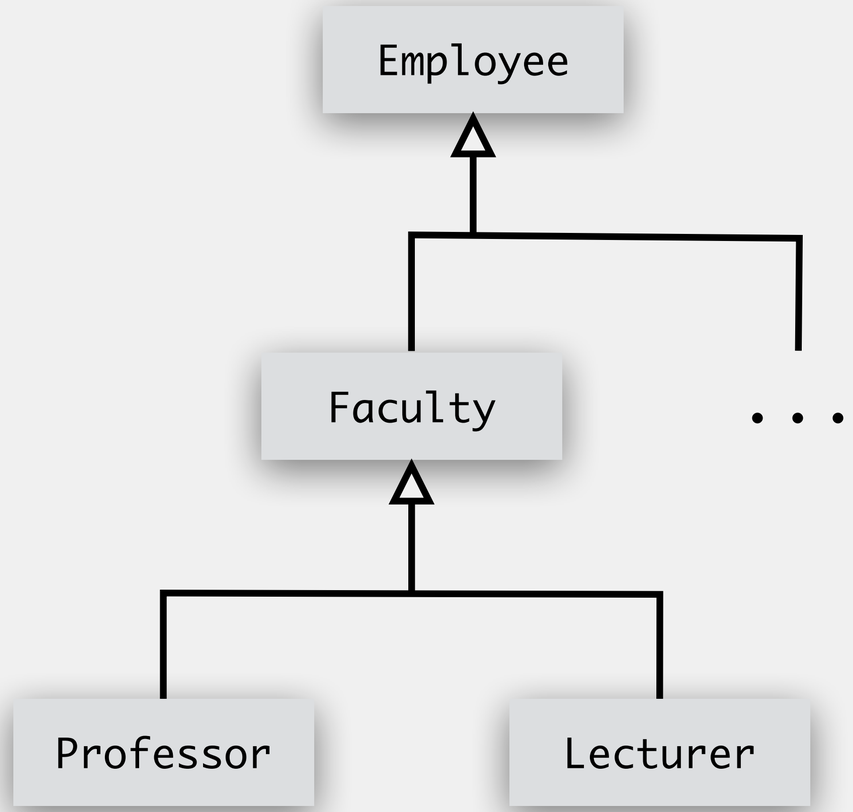
A Hierarchy of Classes

- Things in the real world are often grouped together, or share characteristics
  - dogs, cats, and horses are all mammals; mammals, fish, and birds are all animals
- Object oriented programming models the real world
  - thus, we should model these relationships
- *inheritance*: specifying commonalities/differences between related classes
  - commonalities in *superclass* (*parent*)
  - differences in *subclass* (*child*)

Inheritance Example



Inheritance Example



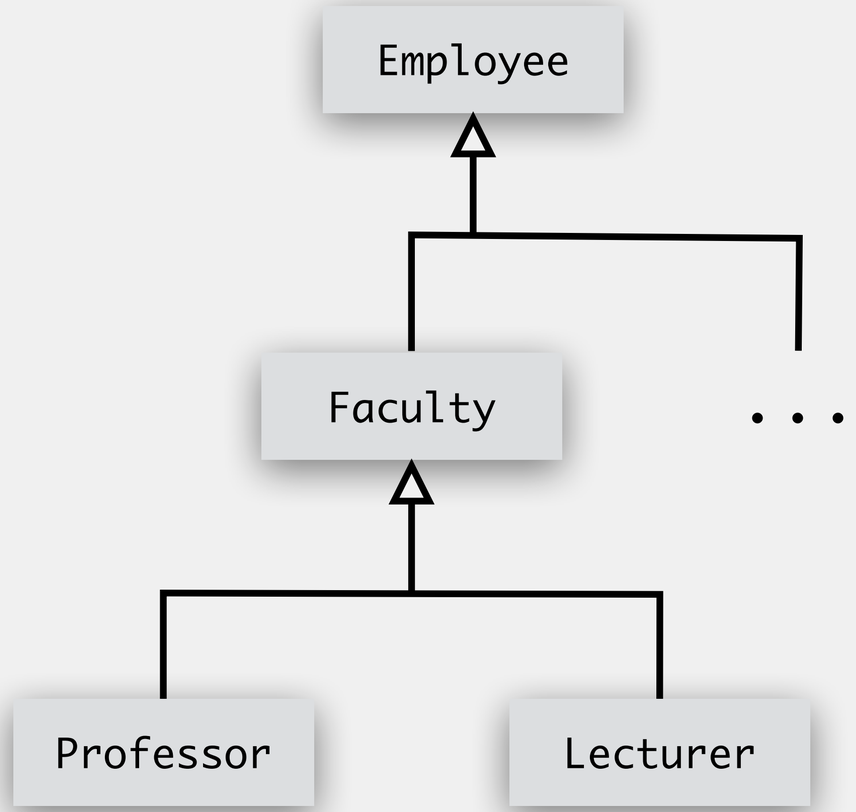

```java
public class Employee {
    ...
}
```

```java
public class Faculty extends Employee {
    ...
}
```

```java
public class Professor extends Faculty {
    ...
}
```

```java
public class Lecturer extends Faculty {
    ...
}
```

Inheritance Example



Child classes can access public/protected attributes/methods in parent classes

- extends up the hierarchy
- e.g., Professor can access Faculty, Employee

Parent classes cannot access **anything** in a child class

- e.g., Employee cannot access anything in Faculty, Professor, Lecturer, ...

Polymorphism

- *Polymorphism* is the occurrence of something in many different forms
  - in the case of programming, methods

- Two types of polymorphism
  - *overriding* occurs when a child class replaces a method from a parent class
  - *overloading* occurs when several methods in a class share the same name but with different parameters

Method Overriding

```
public class Employee {
    ...

    public int getContractLength() {
      return 12;
    }

    ...
}
```

```
public class Faculty extends Employee {
    ...

    public int getContractLength() {
      return 9;
    }

    ...
}
```

Any objects of type `Employee`, or that inherit from `Employee`, will use the method found in `Employee`

> …except for `Faculty` and its subclasses which will *override* the method with the version found in `Faculty`

Methods must have the same signature to override

Method Overloading

```
public class Faculty extends Employee {

    String[] dept = new String[1];

    ...

    public void setDept(String dept) {
      this.dept[0] = dept;
    }

    public void setDept(String[] dept) {
      this.dept = new String[dept.length];
      for (int i = 0; i < dept.length; i++) {
        this.dept[i] = dept[i];
      }
    }

    ...
}
```

Commonly used for constructor method, but can be used for any method

- e.g., `Scanner` can be instantiated with a variety of different input sources, each input source requires its own constructor

Java will determine which version to call based on parameters

Method Overloading

```
public class Faculty extends Employee {

    String[] dept;

    ...

    public void setDept(String dept) {
      this.setDept(new String[]{dept});
    }

    public void setDept(String[] dept) {
      this.dept = new String[dept.length];
      for (int i = 0; i < dept.length; i++) {
        this.dept[i] = dept[i];
      }
    }

    ...
}
```

Can even call from one version of the method to another

- again, Java will determine which version

Notice the use of the `this` keyword to reference the current object!

Method Overloading

```
public class Faculty extends Employee {

    String[] dept;

    ...

    public void setDept(String dept) {
      this.setDept(new String[]{dept});
    }

    public void setDept(String[] dept) {
      this.dept = new String[dept.length];
      for (int i = 0; i < dept.length; i++) {
        this.dept[i] = dept[i];
      }
    }

    ...
}
```

To successfully overload a method, one or more of the following must change:

- the type of the parameter(s)
- the number of parameters
- the order of parameters
  - if of two or more types

Super

- The `this` keyword allows us to refer to an object when we are in its instance
- The `super` keyword allows us to refer to an object's parent
- Can be used just like other method/variable references
  - `super()  // calls the parent's constructor`
  - `super(arg1, arg2, ...)  // calls the parent's constructor`
  - `super.methodName()  // calls a method in the parent class`
  - `super.attributeName  // references a parent's attribute`
- Can omit the `super` in the last two examples if there is not an attribute/ method of the same name in the current class

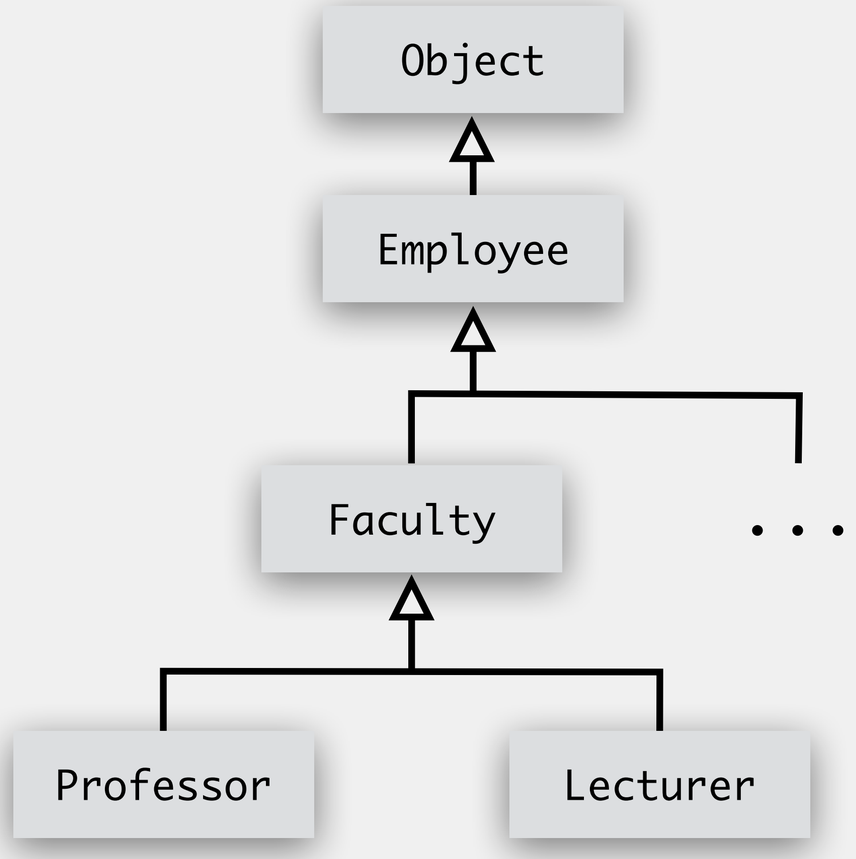
Example: Super

```
public class Employee {
    ...

    public int getContractLength() {
      return 12;
    }

    ...
}
```

```
public class Faculty extends Employee {
    ...

    public int getContractLength() {
      return super.getContractLength() - 3;
    }

    ...
}
```

Will call the parent class, use the returned value from the parent to complete the calculation

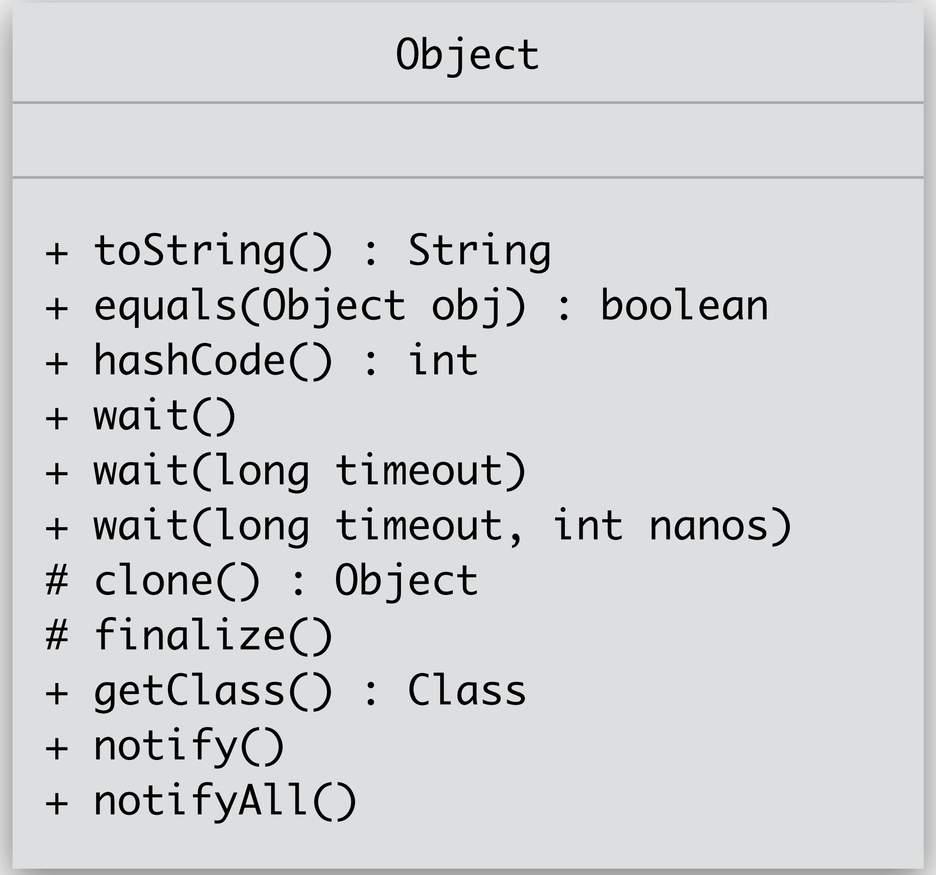
A Hierarchy of Classes



A class without a parent class is automatically a child class of the `Object` class, even if it is not explicitly stated

Thus, every class in Java has the `Object` class as an ancestor

The `Object` Class

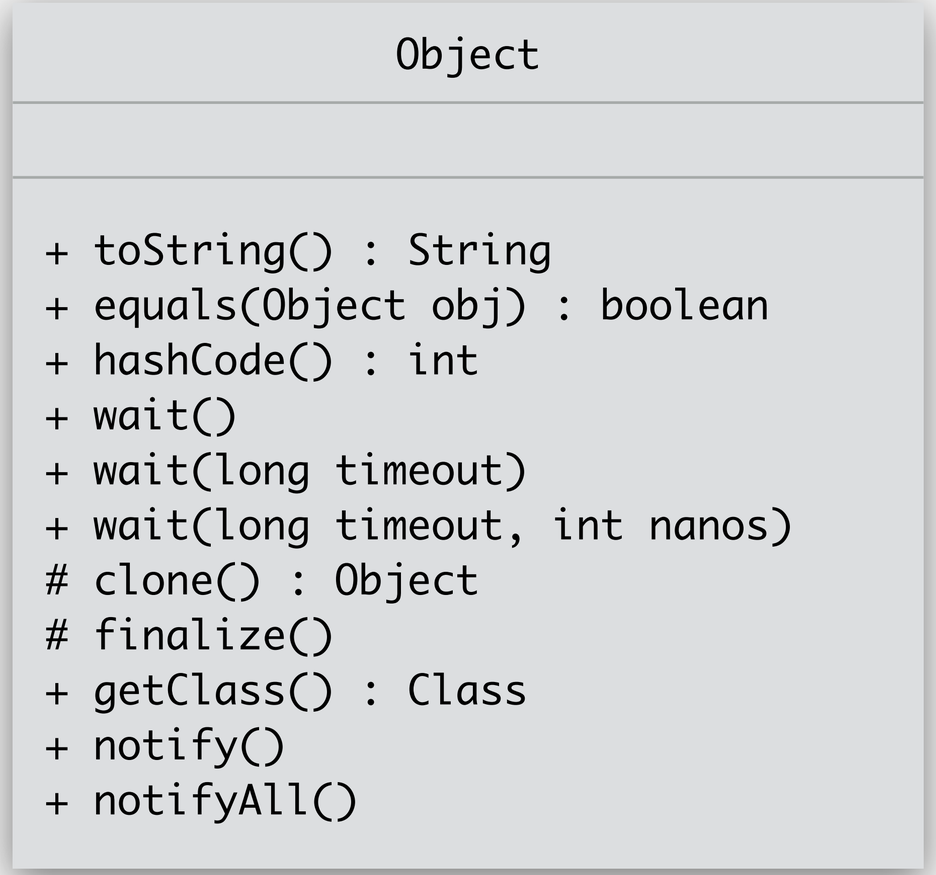


Provides basic implementations of methods critical to using objects

- e.g., providing a text representation of an object
- e.g., checking for equality between two objects

Can override to redefine behavior for a class

The `Object` Class



Note that the `toString()` method is called on any object whenever an object is printed to the console

e.g.,

```
System.out.print(profObject)
```

will actually yield

```
System.out.print(profObject.toString())
```

even if it is not explicitly stated

What is a Type?

Might have heard this term before

- data type, primitive type, class type…

type: a classification for data that tells a programming language how that data can be used

- values of number types can be added, subtracted, multiplied…
- values of `String` type can be concatenated, substring-ed, printed out

Categories of types in Java

- primitive
- interface
- class
- array

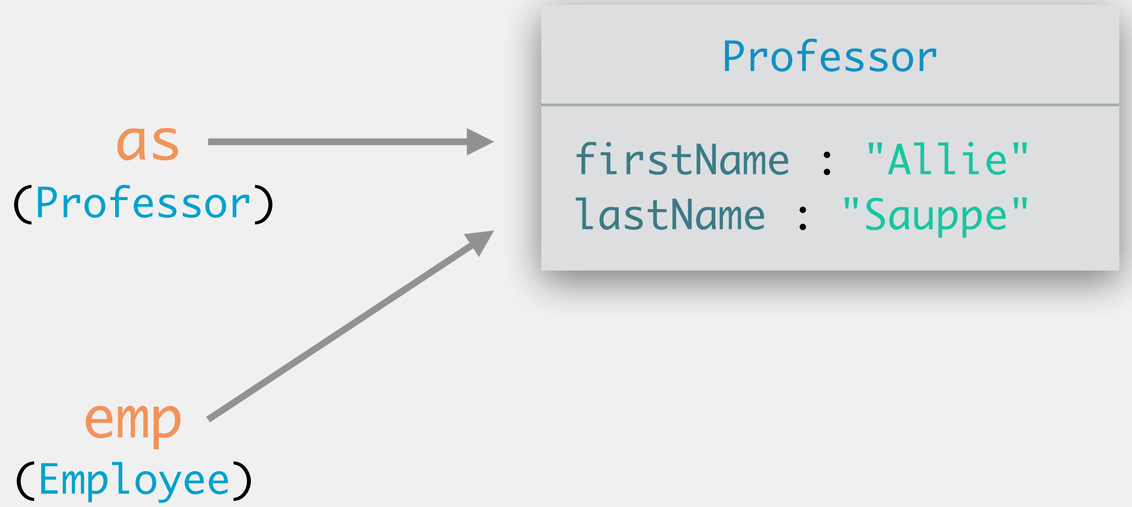
Variables & Type

- Every variable has a type
- Every piece of data (i.e., object) has a type
- Assignment of data to a variable is dependent on the types of each

N.B.: the type associated with the data **must** match or be a subtype of the type of the variable

Type Conformance

```
> Professor as = new Professor("Sauppe", "Allie");

> Employee emp = as;
```



The object stored in a variable might have a type different than the variable itself

- i.e., an object can take on several different guises

What can we do with the object?

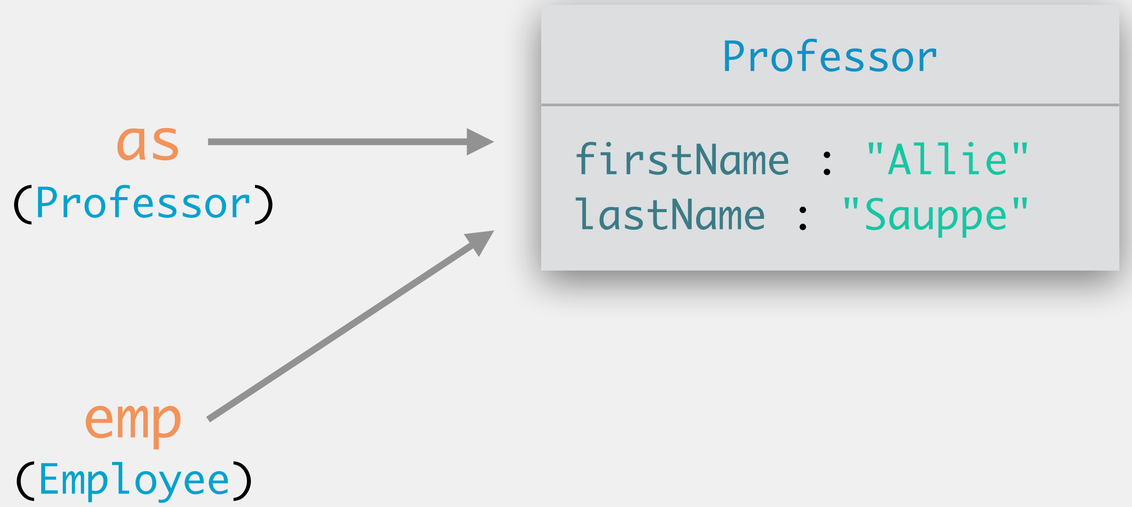
- depends on the type of the variable

How will the object behave?

- depends on the type of the object

Type Conformance

```
Professor as = new Professor("Sauppe", "Allie");

Employee emp = as;
```



Type conformance is when an object of type X conforms to a variable of type Y

- X must be the same as or a subclass of Y

Ask yourself: does the type on the right of the = conform to the type on the left?

- i.e., is the type on the right a descendant of the type on the left?

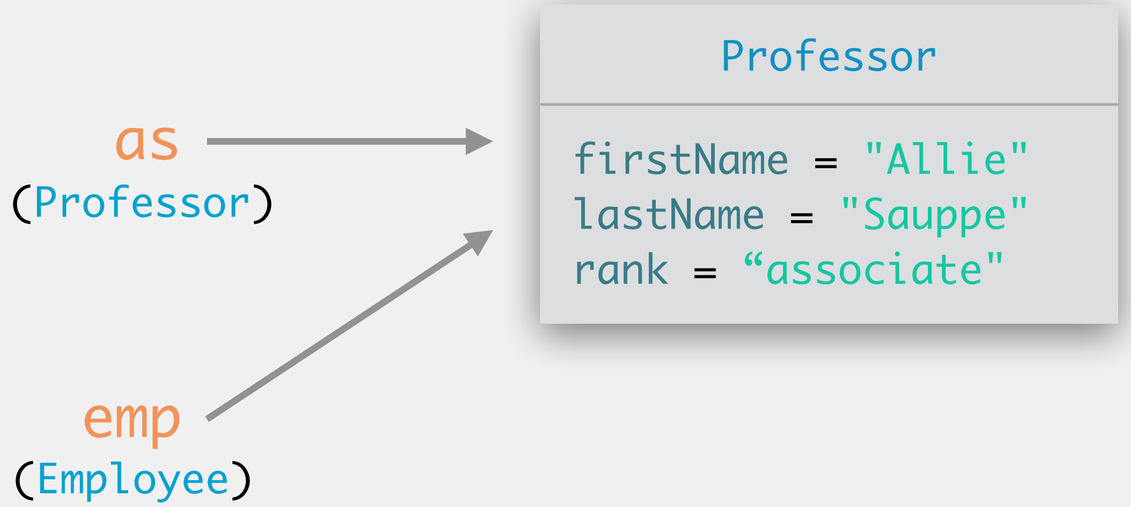
Subtype Polymorphism

```
public class Employee {
    ...
    public String toString() {
      return firstName + " " + lastName;
    }
}
```

```
public class Professor extends Faculty {
    ...
    public String toString() {
      return super.toString() + ", " + rank;
    }
}
```

```
Professor as = new Professor("Sauppe", "Allie");
Employee emp = as;
System.out.println(as);
System.out.println(emp);
```

Subtype polymorphism ensures that an object behaves according to its type, rather than the variable's type



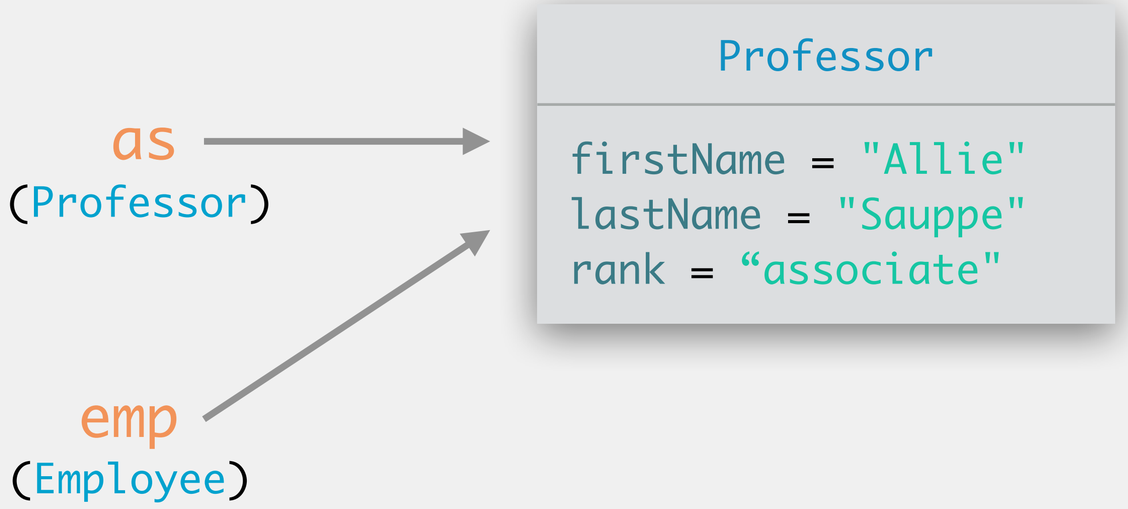
Subtype Polymorphism

```
public class Employee {
    ...
    public String toString() {
      return firstName + " " + lastName;
    }
}
```

```
public class Professor extends Faculty {
    ...
    public String toString() {
      return super.toString() + ", " + rank;
    }
}
```

```
Professor as = new Professor("Sauppe", "Allie");
Employee emp = as;
System.out.println(as);
System.out.println(emp);
```

```
Allie Sauppe, associate
Allie Sauppe, associate
```



Groups of Related Objects

```
Employee[] emps = new Employee[3];
emps[0] = new Professor("Sauppe", "Allie");
emps[1] = new Security("Smith", "John");
emps[2] = new ADA("Yoshizumi", “Becky");

for (int i = 0; i < emps.length; i++) {
    System.out.println(emps[i]);
}
```

```
// prints Allie Sauppe according to the Professor class
// prints John Smith according to the Security class
// prints Becky Yoshizumi according to the ADA class
```

Type conformance allows us to store multiple, related objects together in a single data structure

Subtype polymorphism ensures that an object behaves according to its type, rather than the variable’s type

Subtype Polymorphism

Java has two steps to get from code to execution

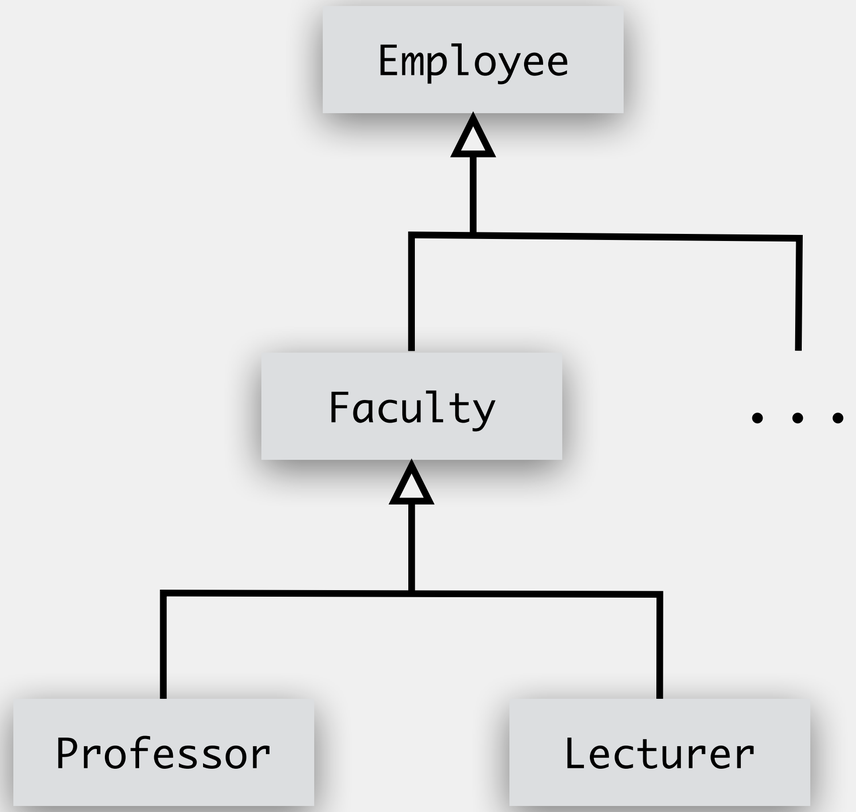
- *compile time* is when Java checks to make sure your code is syntactically valid
  - Java does not yet know the values of variables
  - produces an intermediate form of your program known as Java bytecode (i.e., .class files)
  - this is constantly happening in the background in Eclipse; it is what produces the red underlines

- *run time* is when Java executes your program
  - Java now knows what the values of the variables are!
  - can use variables according to the type of the object stored

Advantages of Inheritance

- Can reuse code
  - write a method once, inherit from the class
  - good to not copy and paste code!
    - see: loops, methods
- Can store objects of related (but different) types in a single data structure
  - data structures store only one type of object
  - can use type conformance to store objects of different types that have a common ancestor
    - more on this next week
  - e.g., `Employee[]` can store objects of type `Professor`, `Lecturer`, `ADA` ...

Disadvantage



Might need to hunt for a method definition

Example:

`Employee` might have a method `getName()`

can be difficult to know that if you are looking in the `Professor` class and the method has not been overridden

Programming Style

The **clarity of your code** indicates the **clarity of your thoughts**.

Java Style: Things to Pay Attention To

- Descriptive, appropriate variable names
- Indentation
  - helps to communicate the control flow of your code
  - any reasonable code editor will auto-indent
- Commenting the class (required), methods (required), code (as necessary)
- Writing elegant code
  - i.e., is there a better, more understandable way to write this piece of code?
  - e.g., moving code to a method or loop rather than copying and pasting
  - White space
  - use blank lines (judiciously) to make code more readable

Javadoc Method Comments

```
/**
 * Finds the zero or more courses currently
 * being offered by a particular department.
 *
 * @param dept The prefix code for the dept
 * @return An array of zero or more courses
 *         taught by dept
 */
public Course[] findDeptCourses(String dept) {

   ...
}
```

Accepted convention for formatting comments for a method

- placed above the method
- starts with /**
- description of method
- list of parameters, if any
  - one on each line, in order of appearance
- value returned, if any

Javadoc Method Comments

```
/**
 * Finds the zero or more courses currently
 * being offered by a particular department.
 *
 * @param dept The prefix code for the dept
 * @return An array of zero or more courses
 *         taught by dept
 */
```

You should be able to generate the method signature based on the Javadoc comment