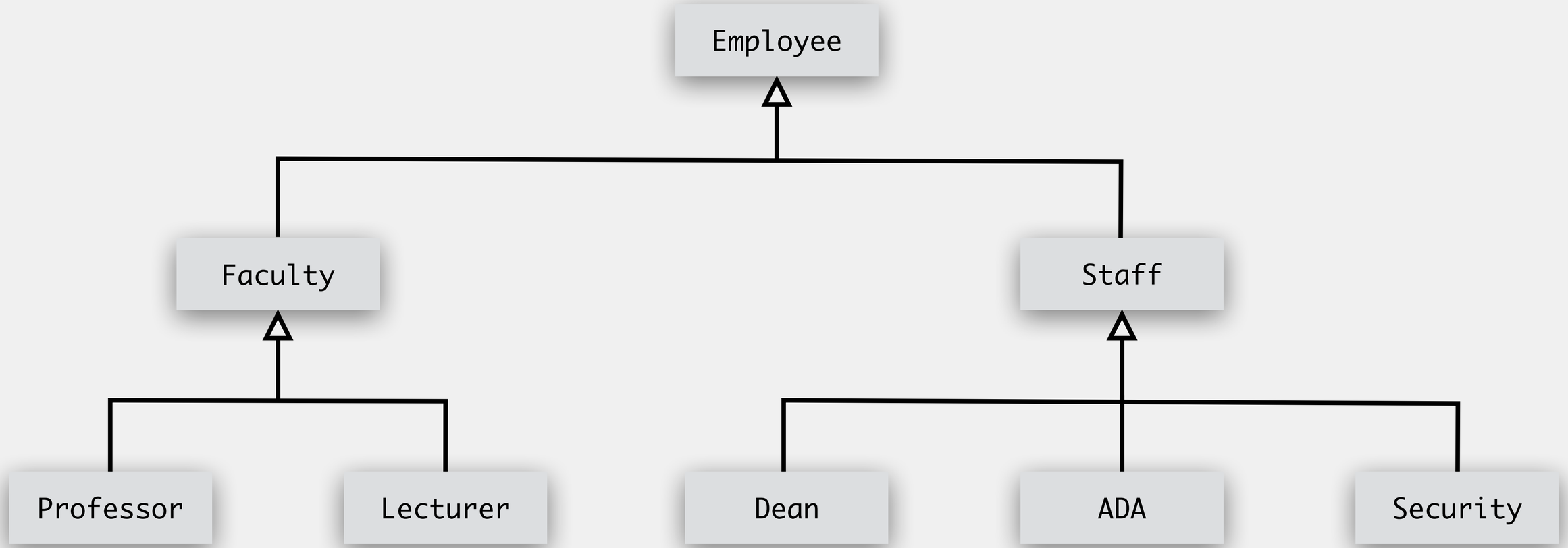
University of Wisconsin La Crosse Computer Science

Week 02:
Abstract Classes & Interfaces

CS 220: Software Design II — D. Mathias

Inheritance Example



Advantages of Inheritance

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 later, but should be familiar from CS 120
- e.g., `Employee[]` can store objects of type `Professor`, `Lecturer`, `ADA` …

But what if what ties these objects together are just abstract traits?

- e.g., employees accrue money towards retirement, but how that is calculated depends on the type of employee (e.g., faculty, staff)

Options

Option 1: provide a default implementation in the parent class, override in the child class(es) as needed

Option 2: provide implementations only in the child classes

Option 3: use *abstract classes*

Can specify that a method is required, but defer implementation to a child class

Options

```
public class Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 12 * 0.05;
    }
}
```

```
public class Faculty extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 9 * 0.05;
    }
}
```

Option 1: provide a default implementation in the parent class, override in the child class(es) as needed

Problem: now have a default implementation that is meaningless

- i.e., what does it mean to just be an employee?

Options

```
public class Employee {
    // no method calcRetirement()
}
```

```
public class Faculty extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 9 * 0.05;
    }
}
```

```
public class Staff extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 12 * 0.05;
    }
}
```

Option 2: provide implementations only in the child classes

Problem: can no longer leverage the benefits of inheritance/type conformance: if we have an array. of objects that conform to Employee, the array type is Employee[] but we can’t apply calcRetirement method to a Faculty object in the array because it doesn’t exist for Employee.

Options

```
public class Employee {
    // no method calcRetirement()
}
```

```
public class Faculty extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 9 * 0.05;
    }
}
```

```
public class Staff extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 12 * 0.05;
    }
}
```

Option 2: provide implementations only in the child classes

this will **not** work without casting, as `Employee` does not have a `calcRetirement()` method

```
Employee[] allEmps;
// allEmps filled with all Employee objects
for (int i = 0; i < allEmps.length; i++) {
  total += allEmps[i].calcRetirement());
}
```

Options

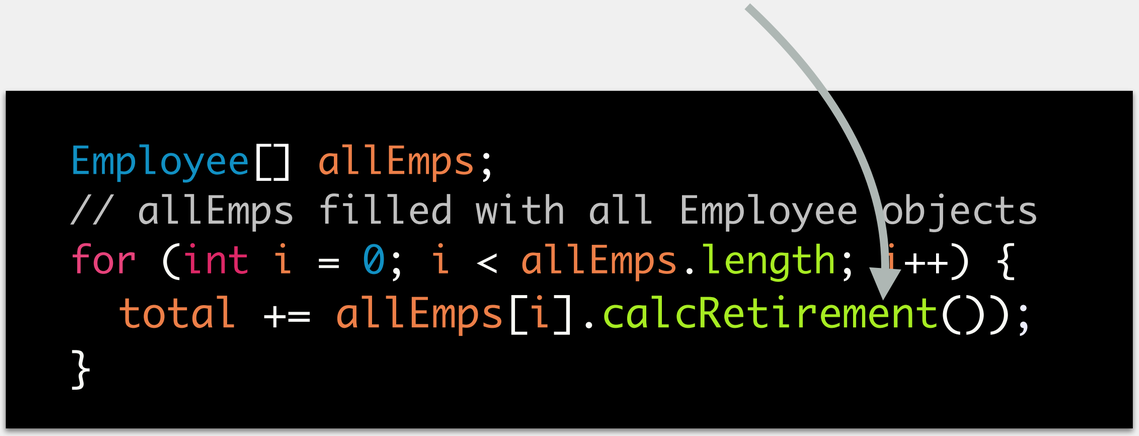
```
public class Employee {
    // no method calcRetirement()
}
```

```
public class Faculty extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 9 * 0.05;
    }
}
```

```
public class Staff extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 12 * 0.05;
    }
}
```

Option 2: provide implementations only in the child classes

Remember: what we can do with an object depends on the variable; how it behaves depends on the object.



```
Employee[] allEmps;
// allEmps filled with all Employee objects
for (int i = 0; i < allEmps.length; i++) {
  total += allEmps[i].calcRetirement());
}
```

Options

```
public abstract class Employee {
    public abstract double calcRetirement();
}
```

```
public class Faculty extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 9 * 0.05;
    }
}
```

```
public class Staff extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 12 * 0.05;
    }
}
```

Option 3: use *abstract classes*

Can specify that a method is required, but defer implementation to a child class

Options

```
public abstract class Employee {
    public abstract double calcRetirement();
}
```

```
public class Faculty extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 9 * 0.05;
    }
}
```

```
public class Staff extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 12 * 0.05;
    }
}
```

Option 3: use *abstract classes*

Parent class specifies that all child classes should have a particular method signature, but allows the child classes to provide the implementation

No meaningless default method

Options

```
public abstract class Employee {
    public abstract double calcRetirement();
}
```

```
public class Faculty extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 9 * 0.05;
    }
}
```

```
public class Staff extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 12 * 0.05;
    }
}
```

Option 3: use *abstract classes*

Child classes can then provide their own unique implementation according to their needs

Options

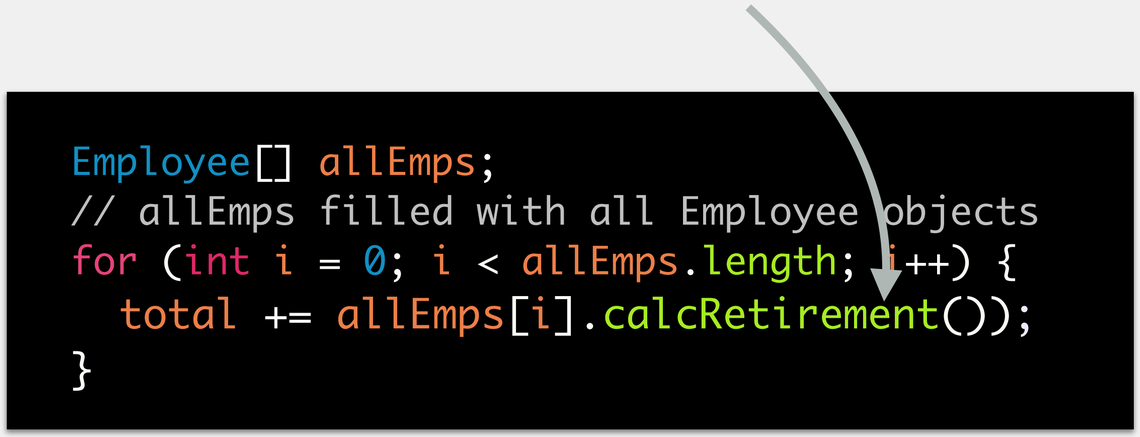
```
public abstract class Employee {
    public abstract double calcRetirement();
}
```

```
public class Faculty extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 9 * 0.05;
    }
}
```

```
public class Staff extends Employee {
    public double calcRetirement() {
      return yrsWrkd * avgSalary / 12 * 0.05;
    }
}
```

Option 3: use *abstract classes*

this **does** work, since every class that extends from Employee is guaranteed to have some implementation of this method



Another Example

Want to have `Shape` class as a superclass to various shapes

- e.g., `Triangle`, `Oval`

All shapes have common properties (e.g., area, perimeter), but calculating them varies by the actual shape we are calculating it for

- i.e., it is meaningless to calculate the area of a `Shape` object

Representing the Abstract

Want to defer implementation of a method to later classes

- may not be defined in the immediate context!

abstract class: a class that defines the shared properties of subclasses without always providing definitions

- zero or more methods can be defined as *abstract*, meaning they have no definition
- zero or more methods can be non-abstract, i.e., they have a definition
  - look just like methods you've seen before

cannot be instantiated

- because some methods are not yet defined

can be subclassed

Purpose of Abstract Classes

- Provide shared attributes and methods for subclasses
- Aide in type conformance
- Allow us to represent something that is a category/group for which the relationship is not tangible

Abstract Classes in Action

```
public abstract class Employee {

    protected String firstName;
    protected String lastName;

    public Employee(String fn, String ln) {
      firstName = fn;
      lastName = ln;
    }

    public String getFullName() {
      return firstName + " " + lastName;
    }

    public abstract double calcRetirement();
}
```

Uses the Java keyword `abstract`

Applied in two places:

- methods without an implementation
- the class signature for the class that contains the abstract method(s)

Class can still have…

- attributes
- non-abstract methods/constructor
- inheritance

Abstract Classes in Action

```
public abstract class Employee {

    protected String firstName;
    protected String lastName;

    public Employee(String fn, String ln) {
      firstName = fn;
      lastName = ln;
    }
    ...
}
```

```
public class Faculty extends Employee {
    public Faculty(String fn, String ln) {
      super(fn, ln);
    }
}
```

Constructor **cannot** be used to instantiate a new abstract object

the following is incorrect syntax

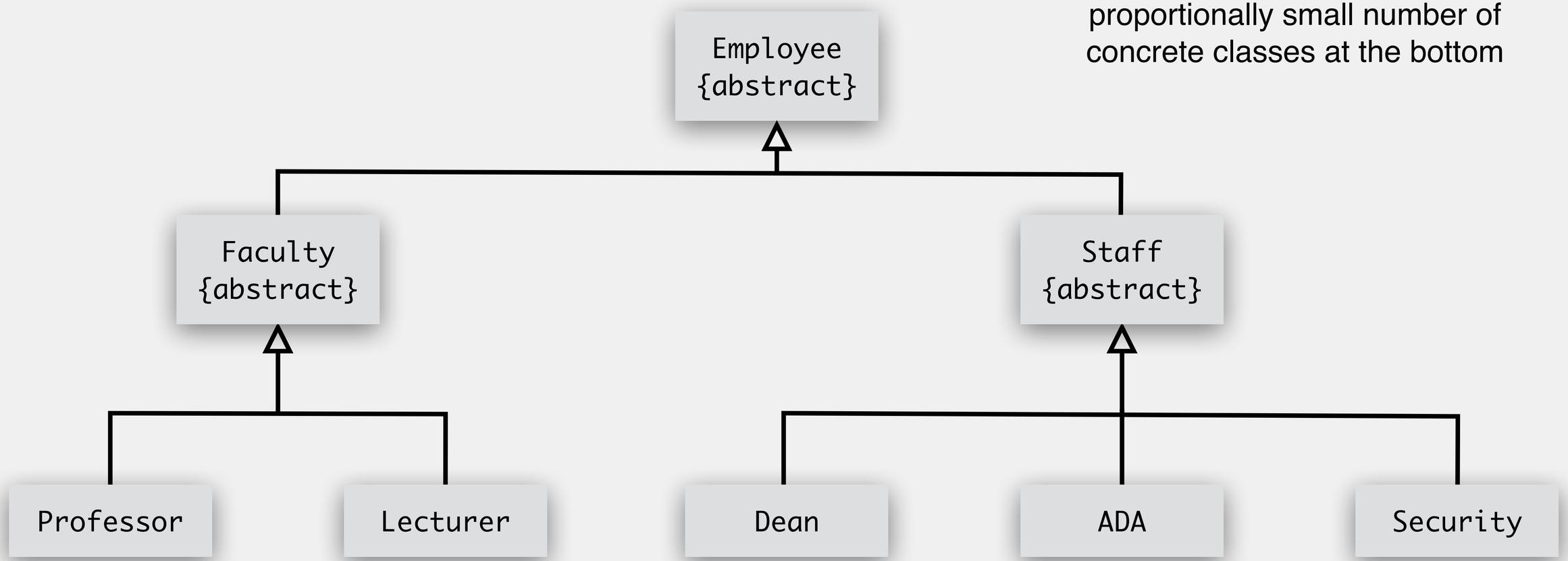
```
Employee e = new Employee(“Tim”, “Smith”);
```

Constructor is only used by child classes

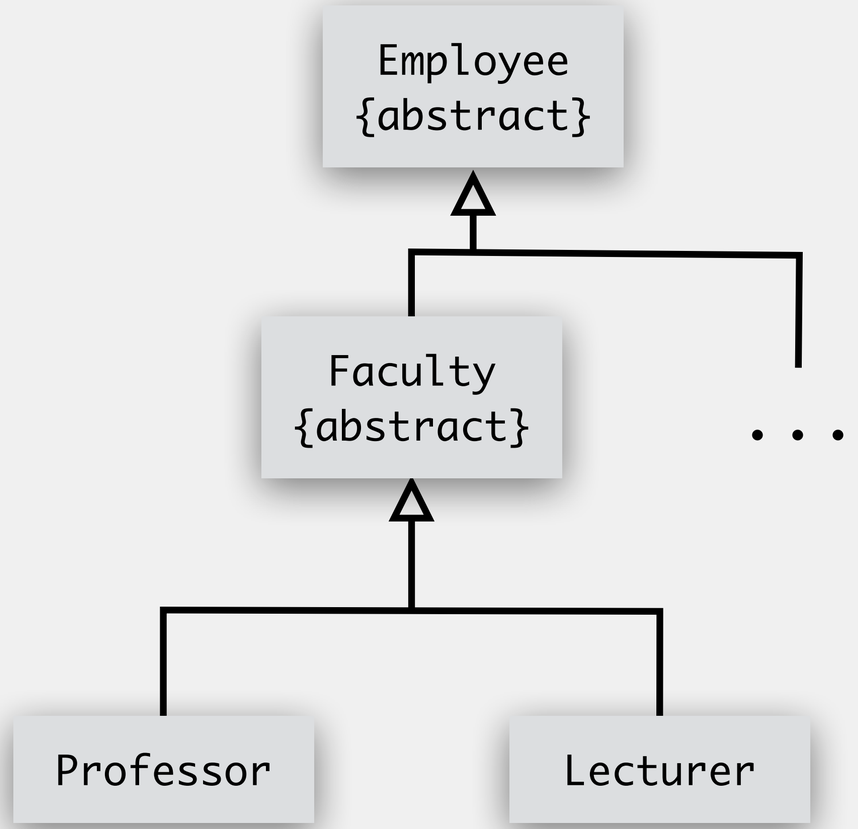
sets attributes shared across child classes

Inheritance Example

Java is made up of many abstract classes at the top levels, and only a proportionally small number of concrete classes at the bottom



Inheritance Example



Child classes can either be abstract or not

- abstract classes can provide implementations for zero or more of the inherited abstract methods
  - can also add new abstract methods
- non-abstract classes must provide implementations for **all** inherited abstract methods
  - `Faculty` didn't provide an implementation from `Employee`? `Professor` and `Lecturer` must
  - `Faculty` did provide an implementation from `Employee`? `Professor` and `Lecturer` can inherit that, or can override it

Exercise

Write the abstract class Staff, which extends Employee

- add a new int attribute vacationTime, which is set in the constructor
- add a new public abstract method calculateVacationAccrual(), which returns double

Write the class ADA, which extends Staff

- the constructor should automatically set the vacation time to 15
- you may choose how to implement the abstract method

Inheritance Revisited

Tying classes together with inheritance establishes an *is-a* relationship

- a professor *is-a* faculty member *is-a* employee…

Sometimes, we want to tie classes together that are not so closely related

- these classes have shared capabilities, but might not be directly related

An Outlandish Example (to illustrate the point)

You are in the middle of a large room when a zombie enters. There are no items within your reach. You yell to a friend near several items: “Quick! Throw me something I can injure the zombie with!”

What are the desired properties of an item your friend should throw you? What are some items that fulfill those properties?

Interfaces

interface: a construct where you can specify what actions a class implementing that interface should be able to take

- no attributes
- no constructor
  - cannot be instantiated, like abstract classes
- no method implementations
  - (except default methods starting in Java 8, but it's a minor use case we're going to skip)

"I need to guarantee that this class can do the following…"

Interfaces in Action

```
public interface ZombieTool {
    public void hurtZombie();
}
```

- Uses the Java keyword `interface`
  - replaces `class` in the class signature
- Just a list of method signatures
  - can be one or more
  - no need to include `abstract`, which is assumed in an interface

Interfaces in Action

```
public interface ZombieTool {
    public void hurtZombie();
}
```

```
public class Shovel implements ZombieTool {
    ...
    public void hurtZombie() {
      System.out.println("Hit zombie!");
    }
}
```

```
public class Dinosaur implements ZombieTool {
    ...
    public void hurtZombie() {
      System.out.println("Eat zombie!");
    }
}
```

Classes that implement this interface have two changes:

- must add `implements` to the class signature
- must provide implementations of **all** methods

Notice how otherwise unrelated `Shovel` and `Dinosaur` are

Interfaces in Action

```
public interface ZombieTool {
    public void hurtZombie();
}
```

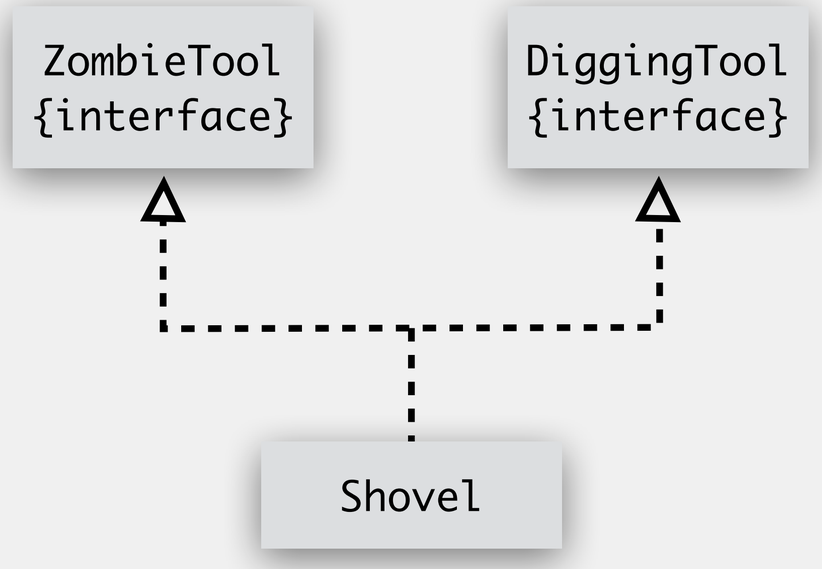
```
public interface DiggingTool {
    public void dig();
    public void dig(int howDeep);
}
```

```
public class Shovel implements ZombieTool, DiggingTool {
    ...
    public void hurtZombie() { System.out.println("Hit zombie!"); }

    public void dig() { System.out.println("Digging..."); }

    public void dig(int howDeep) { System.out.println("Dug "
                                    + howDeep + " feet"); }
}
```

Can even implement from multiple interfaces

Provides a form of multiple inheritance in Java

Interfaces in UML



Representation similar to inheritance

Key differences

- can point to more than one interface
- use of dashed line

Interface Uses

1. Establishing commonly used functionality across disparate classes

   the `compareTo` method is useful for determining order between two objects

   part of the `Comparable` interface

2. Allowing multiple inheritance

   can be dangerous; not covered in CS 220

3. Setting a contract that classes from third-party programmers can use

   not covered in CS 220

Exercise

Write the interface CourseAdmin

- add the method public String getCourses();
- add the method public void addCourses(String course);
- add the method public boolean removeCourse(String course);

Exercise

Now, have both Faculty and ADA implement this interface

What do you think will happen in Faculty regarding method implementation?

Comparable Interface

Comparable {interface}
+ compareTo(Object o) : int

```
Professor as = new Professor("Sauppe", "Allie");
Professor dm = new Professor("Mathias", “David”);
as.compareTo(dm);
```

Compares two objects to determine order

what this means is up to the programmer!

<this>.compareTo(<other>) returns…

0 if the objects are equal

-1 if this object is less than the other object

1 if this object is greater than the other object

Abstract Classes vs Interfaces

Abstract Classes

- allows for attribute/method sharing among related classes (inheritance)
- allow for method conformance among related classes
- allows for some implementation, but not required

Interfaces

- allows for method conformance among unrelated classes
- can enable something akin to multiple inheritance

Which To Choose?

Want to relate related classes? Potentially provide some implementation?

- abstract class

What to guarantee that otherwise-unrelated classes share a particular functionality? No need to provide an implementation?

- interfaces

In reality, it’s a bit of an art form

- I’ll help you decide (or sometimes tell you which to use)

Revisiting Subtypes

Abstract classes and interfaces are also considered potential types

Any class that extends or implements these is considered a subtype

- and thus, you can store an object of that type into a variable of the abstract class/ interface supertype