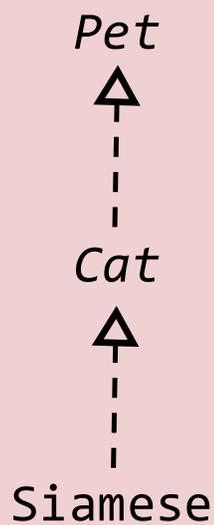# Poll Everywhere

PollEv.com/javabear   text **javabear** to **22333**

Suppose that `Cat <: Pet` and `Siamese <: Cat`. What will be happen if we try to compile and run the bottom code snippet?

Pet

⇡

Cat

⇡

Siamese

```
interface Pet { ... }

interface Cat extends Pet { ... }

class Siamese implements Cat { ... }
```

```
1. Siamese s = new Siamese();
2. Cat c = new Siamese();
3. Pet p = c;
4. s = c;
```

Compiler error (Line 2)   **(A)**

Compiler error (Line 3)   **(B)**

Compiler error (Line 4)   **(C)**

Runs without error   **(D)**

1

Reminders

- A4 due Wednesday

- Prelim 1 in ~ 2.5 weeks
  - Fill out conflict form if necessary (pinned on Ed)
  - More exam info (+ practice exam) coming in ~ 1 week

- TA midterm evaluations later this week (see email)

- Take advantage of support resources (Ed, office hours, assignment resubmissions, etc.)

Reference Type Coercion

Sometimes, the CTRR gets in the way, and we need to adjust the compiler's view to access a behavior

Account savings = new SavingsAccount("Savings", 230000, 3.0);

System.out.println (((SavingsAccount) savings), interestRate() + "%");

We can use casting to adjust the compiler's view
(lower in type hierarchy)

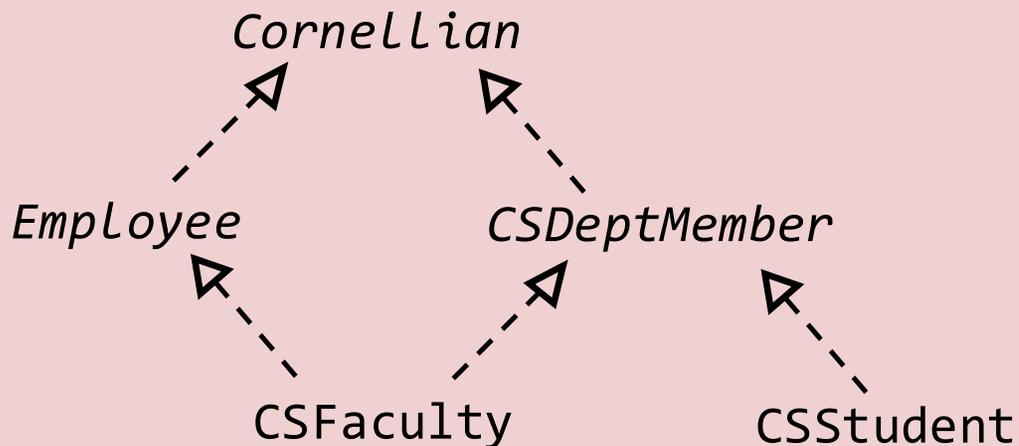Compile Time: Compiler "trusts cast" if it can possibly succeed

Runtime: Dynamic type determines if cast actually works (or if exception is thrown)

Given the following type hierarchy and variable declarations, which cast will **not** compile?

```
Cornellian

Employee          CSDeptMember

CSFaculty              CSStudent
```

Cornellian c;   Employee e;   CSFaculty f;
CSStudent s;   CSDeptMember d;

upcast: not needed, but ok

d = (CSDeptMember) s; **(A)**

works if c → CSFaculty

f = (CSFaculty) c;   **(B)**

works if d → CSFaculty

e = (Employee) d;   **(C)**

no, student can't be Faculty

f = (CSFaculty) s;   **(D)**

Lecture 10: Inheritance

CS 2110

February 24, 2026

Today's Learning Outcomes

43. Describe the principle of dynamic dispatch and the compile-time reference rule.

44. Explain inheritance relationships and their benefits/drawbacks over interfaces.

45. Given a parent class, use inheritance to develop one or more child subclasses.

46. Determine the correct visibility modifier (public, protected, or private) for a given field or method and justify your choice.

47. Trace through the execution of a code sample that includes one or more of the following: inheritance, overridden methods, and super calls

Coding Demo: Our Account Classes

New for today: Checking Accounts may have monthly fee if balance drops too low.

Repetitive "Code Smell"

CheckingAccount

SavingsAccount

Repetition bad for:

- Readability

- Debugging

- Maintainability

- Extensibility

:

Inheritance

We can establish a new kind of subtype relationship, a Subclass relationship, between two classes with the extends keyword.

class B extends A {...}

A (superclass)

↑

B (subclass)

Objects with (dynamic) type B "inherit" the state (fields) and behaviors (methods) from A (and can define their own fields + methods to "specialize" the type)

We can use inheritance to "extract up" common behaviors to a superclass and reduce code duplication.

- "Extract up" common code
- Think about visibility modifiers

Protected Visibility

So far, we've discussed two visibility modifiers:

public = freely visible to every class, the "client interface"

private = visible only within that class, fully <u>encapsulated</u>

↗

guards the class invariant

With inheritance, we'll sometimes want a third option

protected = visible within the class <u>and</u> its subclasses

gives extra tools to subclass developers: "specialization interface"

Good class design requires carefully considering visibility to manage both interactions

Why shouldn't we mark the fields in a superclass as `protected`?

Subtype objects may not have these fields.	**(A)**
Subclasses might not know about these fields' invariants.	**(B)**
Subclasses may want to declare their own fields.	**(C)**
We should... access to the state could be useful in the subclass.	**(D)**

- Identify "specialized" states/behaviors

- Inherit the rest

Method Overrides

When we override a method in the subclass, we <u>replace</u> its definition from the superclass

- same method signature (name, params)

- "shadowed" by dynamic dispatch

- should still conform to superclass specs, but can refine them

- annotate with @Override and write fresh refined specs

The super Keyword

Lets us "dispatch up" to higher method definition

calling super.<method name>() invokes the next higher

up definition of <method name>

Useful to do extra work in override, then refer back to

superclass definition

super(), perhaps with arguments, calls superclass constructor

to set up those inherited fields

- first line of subclass constructor <u>must be</u> superconstructor call

* By default, Java inserts no-argument super() call, If there

isn't a no-arguments constructor, this can't happen

An Encapsulation Problem

```
public class Account {
  /** Reassigns the transaction log to a new `StringBuilder` object that contains this
   *    month's initial account balance. This method must be  called at the start of every
   *    month (i.e., from the constructor and during the execution of `transactionReport()`). */
  private void resetTransactionLog() { … }

  public String transactionReport() { … this.resetTransactionLog(); … }
}
```

If the subclass overrides transaction Report() and
doesn't call super. transactionReport(), this will
break the Account class invariant.

Abstract Classes

Classes and interfaces are two "extremes"

class = complete definition of all methods, instantiable

interface = no method definitions, just signatures, not instantiable

abstract classes are middle ground; offer some complete definitions and declare some abstract methods with defs left as responsibility for subclass

missing method defs = not instantiable

subclass must either
 – provide definitions for all abstract methods
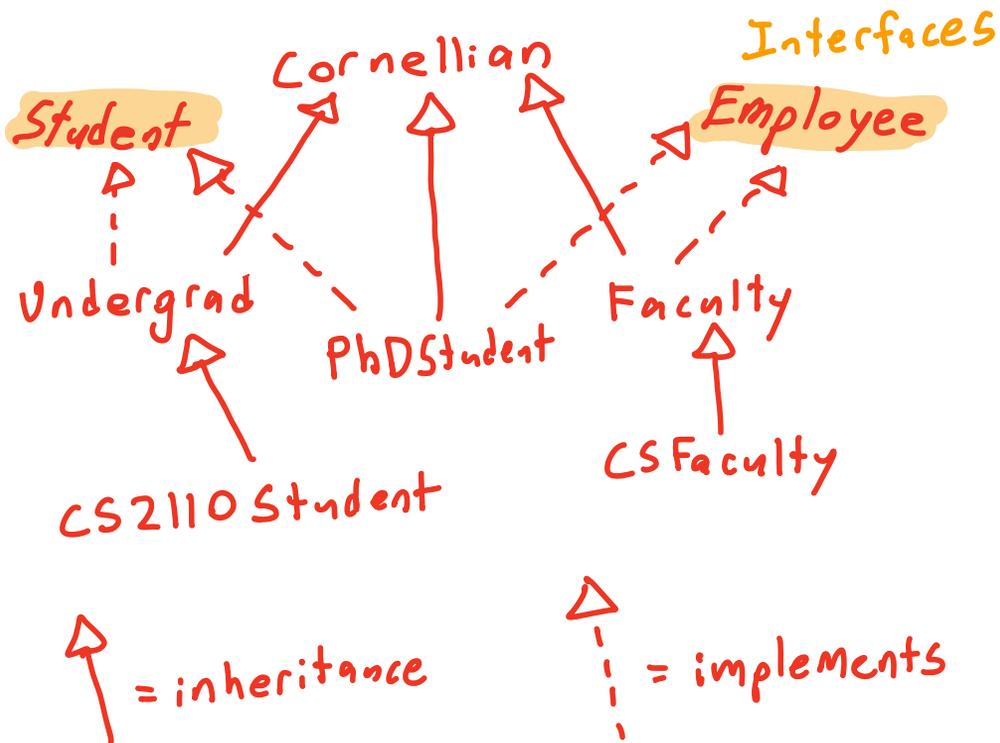 – be abstract itself

- mark transaction Report() as final to prevent subclasses from overriding and breaking Account invariant

- add abstract method call closeOutMonth() to let subclasses define specialized end-of-month things to go on transaction report

Interfaces and inheritance let us build complex, deep type hierarchies

Cornellian
Interfaces
Student
Employee

Undergrad

PhDStudent
Faculty

CS2110 Student

CSFaculty

↑ = inheritance

↑ = implements

Java's single-inheritance rule:

Every* class has exactly one parent class (i.e., immediate superclass)

*choose wisely, this locks you into a rigid relationship

A class can implement any number of interfaces

- interfaces simply promise existence of behaviors

- often prefer interfaces to inheritance, unless there's significant code reuse