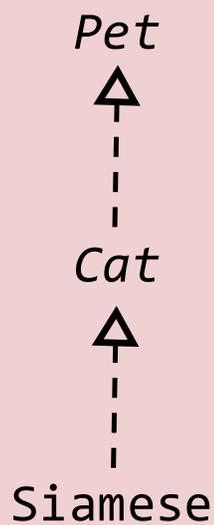


Poll Everywhere

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?



```
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)**

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

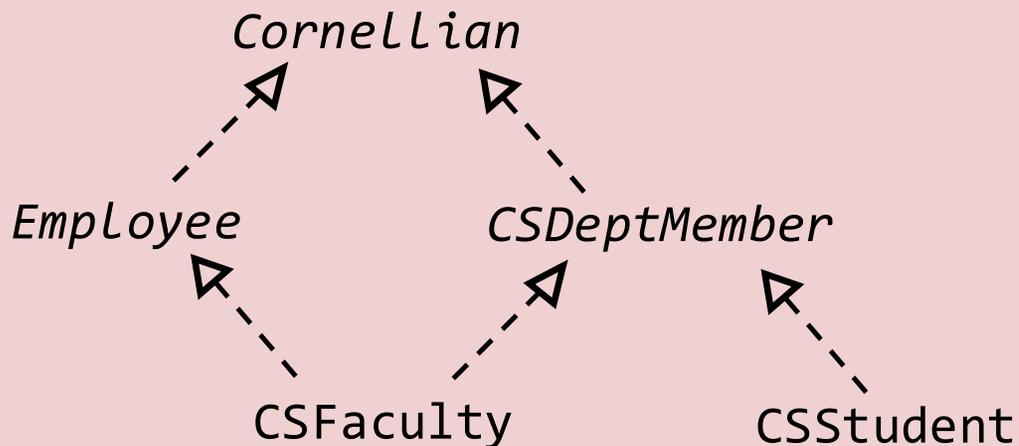
Runtime: Dynamic type determines if cast actually works (or if exception is thrown)

Poll Everywhere

PollEv.com/javabear text javabear to 22333



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



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

```
/** Models a checking account in our personal finance app. */
public class CheckingAccount implements Account {

    /** The balance must remain above the amount, in cents, to prevent
     *  the monthly account fee.
     */
    public static final int MINIMUM_BALANCE = 10000;

    /** The monthly account fee.
     */
    public static final int ACCOUNT_FEE = 500;

    /** The name of this account.
     */
    private String name;

    /** The current balance, in cents, of this account.
     */
    private int balance;

    /** The current month's transaction report
     */
    private StringBuilder transactions;

    /** Whether the account fee should be charged this month.
     */
    private boolean chargeFee;

    /** Constructs a checking account with given name and initial balance.
     */
    public CheckingAccount(String name, int balance) {
        this.name = name;
        this.balance = balance;
        resetTransactionLog();
    }

    /** Reassigns the transaction log to a new StringBuilder object that
     *  contains this month's initial account balance, and resets chargeFee.
     */
    private void resetTransactionLog() {
        this.chargeFee = this.balance < MINIMUM_BALANCE;
        this.transactions = new StringBuilder("Initial Balance:");
        this.transactions.append(centsToLocalizedString(this.balance));
        this.transactions.append("\n"); // newline
    }

    /** Converts the given number of cents into a String in "$X.XX" format
     */
    private static String centsToLocalizedString(int cents) {
        int dollars = cents / 100;
        cents = cents % 100;
        return (" $" + dollars + "." + (cents < 10 ? "0" : "") + cents);
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public int getBalance() {
        return this.balance;
    }

    /** Deposits the specified amount, in cents, to the balance of the
     *  account by logging this transaction with the given memo, and returns
     *  true to indicate that this deposit was successful. Requires that
     *  amount > 0.
     */
    @Override
    public boolean depositFunds(int amount, String memo) {
        assert amount > 0;
        this.balance += amount;
        this.transactions.append("- Deposit ");
        this.transactions.append(centsToLocalizedString(amount));
        this.transactions.append("\n");
        this.transactions.append(memo);
        this.transactions.append("\n");
        return true; // we can always add funds to a checking account
    }

    /** Attempts to transfer the specified amount, in cents, from this account
     *  to receiveAccount and return whether this transaction was successful.
     *  If this transaction is successful, it is logged to both accounts with the
     *  given memo. Otherwise, no changes are made to either account and no
     *  transaction is logged. Requires chargeFee if balance drops below
     *  MINIMUM_BALANCE. Requires that amount > 0.
     */
    @Override
    public boolean transferFunds(Account receivingAccount, int amount) {
        assert amount > 0;
        if (amount > this.balance)
            return false; // insufficient funds

        if (receivingAccount.depositFunds(amount, "Transfer from " + this.name))
            return false; // could not add funds

        this.balance -= amount;
        this.chargeFee |= this.balance < MINIMUM_BALANCE;
        this.transactions.append("- Transfer ");
        this.transactions.append(centsToLocalizedString(amount));
        this.transactions.append("\n");
        this.transactions.append(receivingAccount.getName());
        this.transactions.append("\n");
        return true;
    }

    /** Called once at the end of each month to return a String summarizing the
     *  account's initial balance that month, all transactions made during that
     *  month, and its final balance. As a final transaction for the month,
     *  a interest is accrued to the account based on the current rate.
     */
    @Override
    public String transactionReport() {
        this.processMonthlyFee();
        this.transactions.append("Final Balance:");
        this.transactions.append(centsToLocalizedString(this.balance));
        this.transactions.append("\n"); // newline

        String report = this.transactions.toString();
        resetTransactionLog();
        return report;
    }

    /** Debits the monthly account fee if the balance fell below the
     *  minimum during the month.
     */
    private void processMonthlyFee() {
        if (this.chargeFee) {
            this.balance -= ACCOUNT_FEE;
            this.transactions.append("- Withdraw ");
            this.transactions.append(centsToLocalizedString(ACCOUNT_FEE));
            this.transactions.append("\n");
            this.transactions.append("Account Maintenance Fee");
            this.transactions.append("\n");
        }
    }
}
```

SavingsAccount

```
/** Models a savings account in our personal finance app.
 *  public class SavingsAccount implements Account {

    /** The name of this account.
     */
    private String name;

    /** The current balance, in cents, of this account.
     */
    private int balance;

    /** The current (nominal) APR of this account.
     */
    private double apr;

    /** The current month's transaction report
     */
    private StringBuilder transactions;

    /** Constructs a savings account with given name, initial balance, an interest rate.
     */
    public SavingsAccount(String name, int balance, double rate) {
        this.name = name;
        this.balance = balance;
        this.apr = rate;
        this.resetTransactionLog();
    }

    /** Reassigns the transaction log to a new StringBuilder object that contains the month's
     *  initial balance, and resets interest.
     */
    private void resetTransactionLog() {
        this.transactions = new StringBuilder("Initial Balance:");
        this.transactions.append(centsToLocalizedString(this.balance));
        this.transactions.append("\n"); // newline
    }

    /** Converts the given number of cents into a String in "$X.XX" format
     */
    private static String centsToLocalizedString(int cents) {
        int dollars = cents / 100;
        cents = cents % 100;
        return (" $" + dollars + "." + (cents < 10 ? "0" : "") + cents);
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public int getBalance() {
        return this.balance;
    }

    /** Returns the current (nominal) APR of this account
     */
    public double getInterestRate() {
        return this.apr;
    }

    /** Deposits the specified amount, in cents, to the balance of the account
     *  by logging this transaction with the given memo, and returns true to indicate
     *  that this deposit was successful. Requires that amount > 0.
     */
    @Override
    public boolean depositFunds(int amount, String memo) {
        assert amount > 0;
        this.balance += amount;
        this.transactions.append("- Deposit ");
        this.transactions.append(centsToLocalizedString(amount));
        this.transactions.append("\n");
        this.transactions.append(memo);
        this.transactions.append("\n");
        return true; // we can always add funds to a savings account
    }

    @Override
    public boolean transferFunds(Account receivingAccount, int amount) {
        assert amount > 0;
        if (amount > this.balance)
            return false; // insufficient funds

        if (receivingAccount.depositFunds(amount, "Transfer from " + this.name))
            return false; // could not add funds

        this.balance -= amount;
        this.transactions.append("- Transfer ");
        this.transactions.append(centsToLocalizedString(amount));
        this.transactions.append("\n");
        this.transactions.append(receivingAccount.getName());
        this.transactions.append("\n");
        return true;
    }

    /** Called once at the end of each month to return a String summarizing the
     *  account's initial balance that month, all transactions made during that
     *  month, and its final balance. As a final transaction for the month,
     *  a interest is accrued to the account based on the current rate.
     */
    @Override
    public String transactionReport() {
        this accrueMonthlyInterest();
        this.transactions.append("Final Balance:");
        this.transactions.append(centsToLocalizedString(this.balance));
        this.transactions.append("\n"); // newline

        String report = this.transactions.toString();
        this.resetTransactionLog();
        return report;
    }

    /** Adds the monthly interest payment to the account balance.
     */
    private void accrueMonthlyInterest() {
        int interestAmount = (int) (this.balance * (this.apr / 12 * 100));
        this.depositFunds(interestAmount, "Monthly interest @ " + this.apr + "%");
    }
}
```

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) keyword.



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.



Coding Demo: The Account Superclass



- "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 encapsulated

guards the class [↑]invariant

With inheritance, we'll sometimes want a third option

protected = visible within the class and its subclasses

gives extra tools to subclass developers: "specialization interface"

Good class design requires carefully considering visibility to manage both interactions

Poll Everywhere

PollEv.com/javabear text `javabear` to 22333



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)**



Coding Demo: The Account Subclasses



- Identify "specialized" states / behaviors
- Inherit the rest

Method Overrides

When we override a method in the subclass, we replace 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 must be superclass constructor 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 `transactionReport()` and doesn't call `super.transactionReport()`, this will break the `Account` class invariant.

..
mm

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



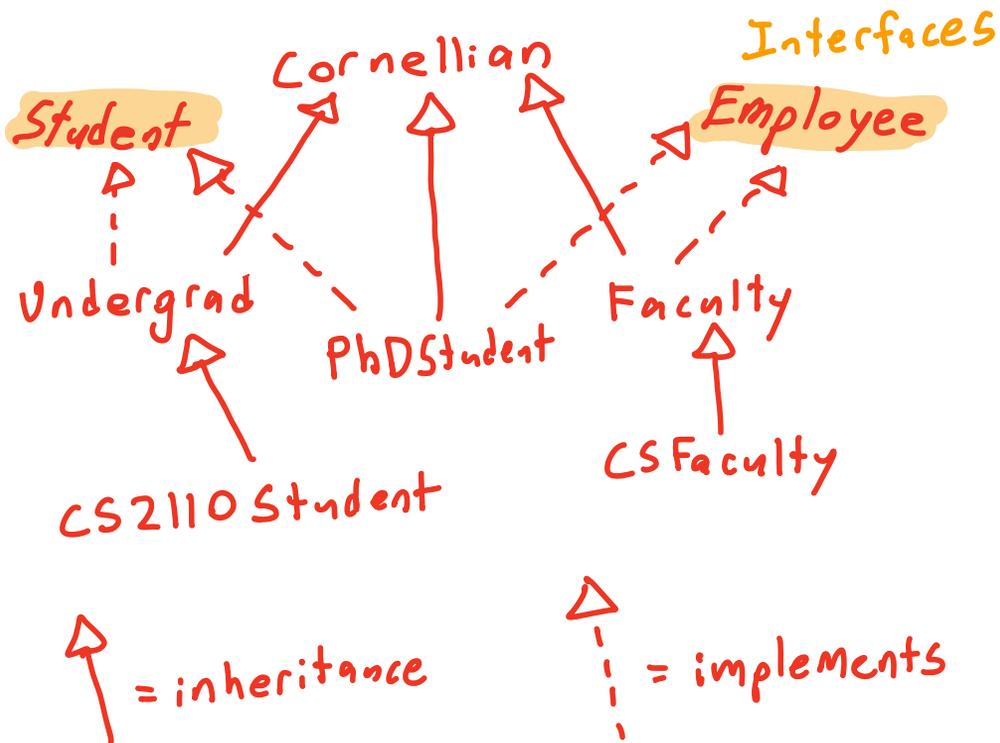
Coding Demo: One More Refactoring



- 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

Type Hierarchies and Single Inheritance

Interfaces and inheritance let us build complex, deep type hierarchies



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