

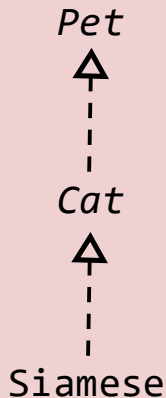
Poll Everywhere

PollEv.com/javabear

text **javabear** to 22333



Suppose that `Cat <: Pet` and `Siamese <: Cat`.
What will 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; ✗ assignment
```

subtype substitution

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

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

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

Runs without error **(D)**

Reference Type Coercion

Sometimes, the **Compile-Time Reference Rule** 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() + "%");
```

Casting can allow us to adjust the compiler's view, but it's a bit subtle. Casts can fail in different ways at different times.

At **compile-time**: Compiler "trusts cast" if it can possibly succeed

At **runtime**: dynamic type determines if cast actually works
otherwise `ClassCastException` is thrown

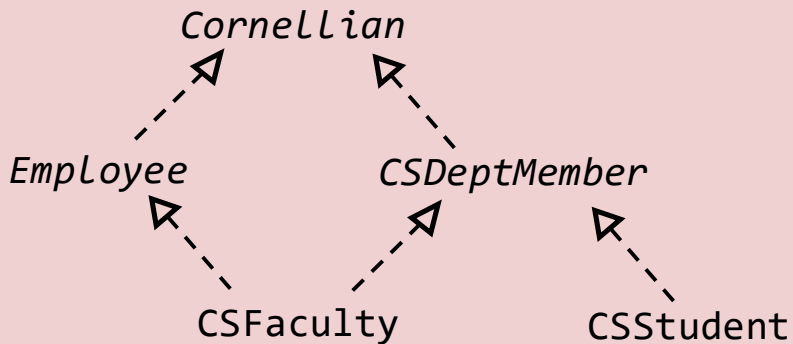
Poll Everywhere

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

September 24, 2026

Today's Learning Outcomes

- 40. Describe the principle of dynamic dispatch and the compile-time reference rule.
- 41. Explain inheritance relationships and their benefits/drawbacks over interfaces.
- 42. Given a parent class, use inheritance to develop one or more child subclasses.
- 43. Determine the correct visibility modifier (public, protected, or private) for a given field or method and justify your choice.
- 44. Suggest and/or carry out a refactoring on a piece of code, and explain how it improves the code's maintainability.



Coding Demo: Our Account Classes



Adjustment from last lecture: CheckingAccounts have a monthly fee if their balance drops too low.

Repetitive "Code Smell"

CheckingAccount

```
/* Models a checking account in our persona/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 the account. */
    private long balance;

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

    /* Whether the account fee should be charged the 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;
        resetTransaction();
    }

    /*
     * 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.transaction = new StringBuilder("Initial Balance: ");
        this.transaction.appendcentsToString(this.balance);
        this.transaction.append("\n");
    }

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

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

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

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

    /*
     * Attempts to transfer the specified 'amount', in cents, from the account
     * to 'receivingAccount', and return whether the transaction was successful.
     * If this transaction is successful, it is a deposit to both accounts with the
     * given 'memo'. Otherwise, no changes are made to either account and the
     * transaction only logs. Updates '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.transaction.append("Transfer ");
        this.transaction.appendcentsToString(amount);
        this.transaction.append("\n");
        this.transaction.append(receivingAccount.name());
        this.transaction.append("\n");
        return true;
    }

    /*
     * Called once at the end of each month to return a 'String' summarizing the
     * account's initial balance that reflects all transactions made during the
     * month, and its final balance. As a final transaction for the month, a
     * maintenance fee is applied to the account if the balance was ever below
     * 'MINIMUM_BALANCE' during the month.
     */
    @Override
    public String transactionReport() {
        this.processMonthlyFee();
        this.transaction.append("Final Balance: ");
        this.transaction.appendcentsToString(this.balance);
        this.transaction.append("\n");

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

    /*
     * Debts 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.transaction.append("Withdraw ");
            this.transaction.appendcentsToString(ACCOUNT_FEE);
            this.transaction.append("\n");
            this.transaction.append("Account Maintenance Fee");
            this.transaction.append("\n");
        }
    }
}
```

SavingsAccount

```
/* Models a savings account in our persona/finance app. */
public class SavingsAccount implements Account {
    /* The name of this account. */
    private String name;

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

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

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

    /*
     * Constructs a savings account with given 'name', 'initial' balance, and interest 'rate'.
     */
    public SavingsAccount(String name, int balance, double rate) {
        this.name = name;
        this.balance = balance;
        this.rate = rate;
        this.transaction = new StringBuilder();
    }

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

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

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

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

    /*
     * Returns the current (nominal) APR for this account.
     */
    public double interestRate() {
        return this.rate;
    }

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

Repetition bad for:

- Readability

- Debugging

- Maintainability

- Extensibility

:

Inheritance

We can establish a new kind of subtype relationship, a **subclass** relationship, between two classes using the `extends` keyword.

```
class B extends A {...}
```

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)

A (superclass)



B (subclass)

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



Coding Demo: The Account Superclass



- "Extract up" code that is common to all Account types
- Think about visibility modifiers

Protected Visibility

Recall: Last week, we introduced two visibility modifiers

public: freely visible to every class, the "**client interface**"

private: visible only within the declaring class, fully **encapsulated**

With inheritance, we 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
- Look for other *"code smells"*

Method Overrides and Dynamic Dispatch

When we override a method in a subclass, we **replace** its definition from the superclass.

- must have same method signature (name, parameters)
- must conform to superclass specs (and likely refine them)

Dynamic dispatch resolves "bottom-up" for each method call, beginning at dynamic type and working upward until suitable method signature is found

- overridden methods "shadow" superclass definitions
- annotate with `@Override` and write fresh refined specs

Poll Everywhere

PollEv.com/javabear

text **javabear** to 22333



```
class A {  
    public f() { System.out.print("1"); }  
}  
  
class B {  
    public g() { System.out.print("2"); f(); }  
}  
  
class C {  
    @Override  
    public f() { System.out.print("3"); }  
}
```

Given the class definitions on the left, what will be printed if we (try to) run the code shown below?

```
B bob = new C();  
bob.g();
```

21

(A)

There will be a compiler error.

(C)

23

(B)

There will be a runtime error.

(D)

Poll Everywhere

PollEv.com/javabear

text javabear to 22333



```
class A {  
    public void f() { System.out.print("1"); }  
}  
  
class B extends A {  
    public void g() { System.out.print("2"); f(); }  
}  
  
class C extends B {  
    @Override  
    public void f() { System.out.print("3"); }  
}
```

① dispatch up to locate inherited g() def

② back down to dynamic type to find f() def

Given the class definitions on the left, what will be printed if we (try to) run the code shown below?

```
B bob = new C();
```

```
bob.g();
```

✓ C <: B
✓ B declares g()

21

(A)

There will be a compiler error.

(C)

23

(B)

There will be a runtime error.

(D)

The super Keyword

Sometimes, we'd like to "dispatch up" to a superclass method definition.

Calling `super.<method name>()` invokes the next definition of `<method name>` up in the type hierarchy.

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 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 `transactionReport()` and doesn't call `super.transactionReport()`, the `Account` class invariant breaks.

Partial solution: mark `transactionReport()` as final
to prevent subclasses from overriding it

but... subclasses refined its behavior...

Abstract Classes

Classes and interfaces are two "extremes"

class: complete definitions of all methods/fields, instantiable

interface: no method definitions (just signatures), no fields, not instantiable

abstract classes are a 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: Our Final Refactoring



Common inheritance pattern:

mark a method as final and then add an abstract
method callback within its definition

"template method pattern"

* we'll see again soon for testing ADTs