

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(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**:

At **runtime**:



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 personaffinance 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 int balance;

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

    /* 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;
        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"); // newline
    }

    /*
     * 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' is successful,
     * true, to indicate that the deposit was successful. Requires that
     * 'amount' > 0.
     */
    @Override
    public boolean deposit(FundAmount, 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 isn't logged. Updates 'chargeFee' if 'balance' drops below
     * MINIMUM_BALANCE. Requires that 'amount' > 0.
     */
    @Override
    public boolean transfer(FundsAccount 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 month, all transactions made during that
     * month, and its final balance. As a final transaction for the month,
     * if interest is accrued to the account based on the current 'rate'.
     */
    @Override
    public String transactionReport() {
        this accrueMonthlyInterest();
        this.transaction.append("Final Balance: ");
        this.transaction.appendcentsToString(this.balance);
        this.transaction.append("\n");

        String report = this.transaction.toString();
        this.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("\nAccount Maintenance Fee");
            this.transaction.append("\n");
        }
    }
}
```

SavingsAccount

```
/* Models a savings account in our personaffinance 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 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.resetTransactionLog();
    }

    /* 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"); // newline
    }

    /* 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 deposit(FundAmount, 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
    }

    @Override
    public boolean transfer(FundsAccount receivingAccount, int amount) {
        public boolean transfer(FundsAccount 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.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 month, all transactions made during that
         * month, and its final balance. As a final transaction for the month,
         * if interest is accrued to the account based on the current 'rate'.
         */
        @Override
        public String transactionReport() {
            this accrueMonthlyInterest();
            this.transaction.append("Final Balance: ");
            this.transaction.appendcentsToString(this.balance);
            this.transaction.append("\n");

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

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

Repetition bad for:

Inheritance

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



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`:



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)

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.

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.

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:



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

Type Hierarchies and Single Inheritance

Interfaces and inheritance let us build deep, complex type hierarchies

Java's single-inheritance rule:

