

CS 351

Design of Large Programs

From Design To Code

September 6, 2021

Class Construct

Class is the fundamental programming concept in Java

- fields
- methods
- modifiers
 - public vs private vs protected
 - static
 - final
 - abstract
- Programs are structured in terms of classes
- Objects are instances of classes
- Execution entails the creation and manipulation of objects

Class Definition

A class can be defined in several ways:

- by defining the fields and methods it provides
- by implementing an existing interface
- by extending an existing class

If class *C extends* class *S*

- *C* is called a subclass, derived class, or child class
- *S* is called a superclass, base class, or parent class
- *C* inherits all the fields and methods associated with *S*
- *C* can add fields and methods
- *C* can override methods (and hide fields) but they can still be accessed by referring to super

Inheritance Illustrated

```
Manager jane = new Manager();  
jane.setSalary(120000.0);  
jane.setBonus(50000.0);  
System.out.println(jane.getSalary());
```

What is the intended control flow?

Inheritance Illustrated

```
public class Employee {  
    private double salary;  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
    public double getSalary() {  
        return salary;  
    }  
}
```

```
public class Manager extends Employee {  
    private double bonus = 0;  
    public void setBonus(double amount) {  
        bonus = amount;  
    }  
    public double getSalary() {  
        return salary + bonus;  
    }  
}
```

What is the error?

Inheritance Illustrated

```
public class Employee {  
    private double salary;  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
    public double getSalary() {  
        return salary;  
    }  
}
```

```
public class Manager extends Employee {  
    private double bonus = 0;  
    public void setBonus(double amount) {  
        bonus = amount;  
    }  
    public double getSalary() {  
        return salary + bonus; salary field not visible  
    }  
}
```

What is the error?

Inheritance Illustrated

```
public class Employee {  
    private double salary;  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
    public double getSalary() {  
        return salary;  
    }  
}
```

```
public class Manager extends Employee {  
    private double bonus = 0;  
    public void setBonus(double amount) {  
        bonus = amount;  
    }  
    public double getSalary() {  
        return getSalary() + bonus;  
    }  
}
```

What is the error?

Inheritance Illustrated

```
public class Employee {  
    private double salary;  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
    public double getSalary() {  
        return salary;  
    }  
}
```

```
public class Manager extends Employee {  
    private double bonus = 0;  
    public void setBonus(double amount) {  
        bonus = amount;  
    }  
    public double getSalary() {  
        return getSalary() + bonus; Infinite recursion  
    }  
}
```

What is the error?

Inheritance Illustrated

```
public class Employee {  
    private double salary;  
    public void setSalary(double salary) {  
        this.salary = salary;  
    }  
    public double getSalary() {  
        return salary;  
    }  
}
```

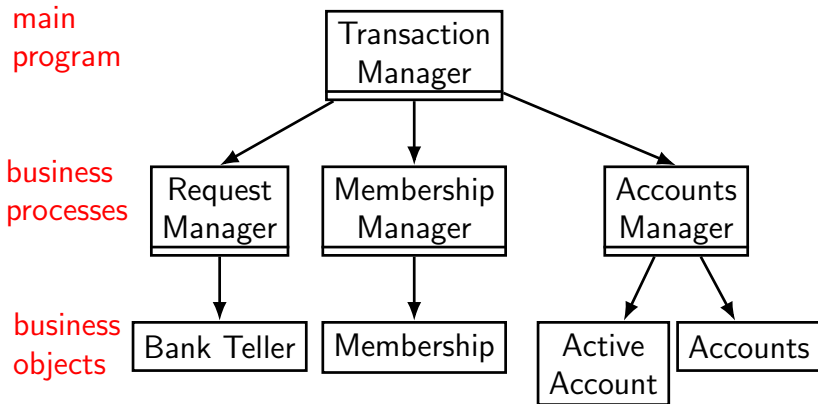
```
public class Manager extends Employee {  
    private double bonus = 0;  
    public void setBonus(double amount) {  
        bonus = amount;  
    }  
    public double getSalary() {  
        return super.getSalary() + bonus;  
    }  
}
```

Case Study: Credit Union

Consider a system that supports the basic operations of a Credit Union:

- manage membership in the credit union by adding and removing individual members
- manage accounts
 - account creation and closing
 - deposit and withdrawal of funds
 - fund transfers among accounts belonging to the same member
- accept and process transaction requests originating with the bank teller

Representative Design Solution



Coding Tasks

1. Define application-specific data types
 - tailored to the application
 - used consistently across the system
 - simplifying the programming task
2. Code procedures and limit access by controlling scope
3. Build classes required to manage the object portfolio

User-Defined Types

Basic application concepts are abstracted as user-defined data types

- ensure continuity with the requirements level
- simplify programming

Such types can vary in complexity

- class with multiple public fields
- class with multiple private fields and extractor methods
- class with private fields and public methods

User-Defined Types: AccountId & TransactionType

AccountId – member account identification

- last name (capitalized, single name)
- account number (9 digits)

```
public class AccountId {  
    private String lastName;  
    private double accountNumber;  
}
```

TransactionType – transaction request type

- create, delete, deposit, withdraw, transfer

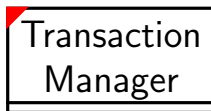
```
public enum TransactionType {  
    CREATE, DELETE, DEPOSIT,  
    WITHDRAW, TRANSFER  
}
```

Main Program

For now: The main program defines the starting point for the entire application

- may or may not be terminating
- controls the sequencing of operations
- may employ local variables for temporary use
- retains little or no information

```
public class TransactionManager {  
    public static void main(String[] args) {  
        // ...  
    }  
}
```



Functional Decomposition

Traditionally, modularity was achieved by means of top-down functional decomposition

- main program embodies the entire functionality of the system
- procedures at levels below decompose it into subfunctions or modules
- clean design enforces a policy of
 - process encapsulation
 - balanced levels of abstraction

In object-oriented design, functional decomposition is present

- at the top layers of the design
- in the design of complex methods

Functional Decomposition: Transaction Manager

```
import managers.RequestManager;
import managers.MembershipManager;
import managers.AccountsManager;

public class TransactionManager {

    public static void main(String[] args) {
        RequestManager requestManager =
            new RequestManager();
        MembershipManager membershipManager =
            new MembershipManager();
        AccountsManager accountsManager =
            new AccountsManager();

        // ...
    }
```

Object Instantiation

A simple way to code an object:

- define the right class
- provide access to the class definition
- instantiate one or more objects as needed

Define class Account inside AccountsManager

- private fields:
 - AccountId (last name, account number—type defined earlier) and balance
- public methods:
 - deposit(amount), debit(amount), balance()
 - accountType() – the last two digits of the account number
- Create one or more instances within the body of the Accounts Manager

Inheritance & Specialization

Objects that relate to each other may still need to be different

Inheritance allows classes to extend a base class

- the latter captures all the common features
- the former adds capabilities specific to an object subtype

Inheritance & Specialization: Account

Accounts can be of two types:

- Checking – they receive a fixed dividend each month
- Savings – they are credited a fixed interest based on the average balance of each month with the daily balance being determined at midnight each day

Inheritance & Specialization: Account

Design changes:

- the Transaction Manager needs access to the system date and time

Subclassing implications:

- Checking requires a new method `payDividend`
- Savings requires
 - a new private field `balanceHistory`
 - a new public method `updateHistory`
 - a new public method `postInterest`

Specialization: Checking & Savings

```
public class Savings extends Account {  
    private float[] balanceHistory;  
    private int day;  
  
    public void updateHistory() {  
        // ...  
    }  
    public void postInterest() {  
        // ...  
    }  
}
```

```
public class Checking extends Account {  
    public void payDividend(float amount) {  
        // ...  
    }  
}
```

Abstract Methods: Account Revisited

The class account may require a method backup

- the actual implementation is account type and system specific

The solution is to define backup as an abstract method

- force Checking and Savings to provide the details
- allow the two subclasses to have different backup approaches, if needed

```
public abstract class Account {  
    public abstract void backup();  
    // ...  
}
```

*No overall design
changes required*

Implementing Interfaces

An interface is a class that provides no implementation for its methods

- no code to execute

A derived class can extend a single base class

- for implementation reasons

A class may implement multiple interfaces

- `MyClass` implements `Interface1`, `Interface2`

Implementing Interfaces: Another Perspective on Account

Some of the requirements on the definition of Account:

- may be related to being a bank account
- may be related to being an insured account

Different requirements can be captured by different interfaces

Implementing Interfaces: Another Perspective on Account

```
public interface BankAccount {  
    // ...  
}
```

```
public interface InsuredAccount {  
    void debitFee();  
    // ...  
}
```

```
public abstract class Account  
    implements BankAccount, InsuredAccount {  
    public abstract void backup();  
    // ...  
}
```

Aggregation & Object Composition

Complex objects are constructed through *object composition*

- the methods of the composite object are coded using methods of nested objects
- these subordinate objects are private
- these subordinate objects should be independent of each other
- the relation between the composite and subordinate objects is called *aggregation*
- this relation between the objects is created by proper composition of classes

Hiding: Minor

One form of specialization entails hiding methods of the superclass

- method is overridden to generate exception if invoked

```
public class MinorSavings extends Savings {  
    @Override  
    public void withdraw(float amount)  
        throws TransactionException {  
        throw new TransactionException("Minors are "  
            + "not permitted to withdraw "  
            + "from savings accounts.");  
    }  
}
```

Designing with Classes

The philosophy of this course:

- focus on design
- exploit language features to realize the design
- keep the design language-independent

Implementation does not require an object-oriented language

A design strategy that is class-centered (often used in practice)

- limits implementation options
- may fail
 - to provide adequate encapsulation
 - to convey clarity of the design