CS202 Java Object Oriented Programming
Encapsulation, Inheritance, and Polymorphism

Chengyu Sun
California State University, Los Angeles

Access Modifiers
- **public** – can be accessed from anywhere
- **private** – can be accessed only within the class
- **protected** – can be accessed within the class, in subclasses, or within the same package
- No modifier – can be accessed within the same package

Access Control Example
```java
public class Foo {
    public int a;
    private int b;
    protected int c;
    public Foo() {
        a = 1;
        b = 2;
        c = 3;
    }
    public void print(Foo f) {
        System.out.println(f.a); // ??
        System.out.println(f.b); // ??
        System.out.println(f.c); // ??
    }
    public static void main(String arg[]) {
        new Foo().print(new Foo());
    }
}
```

Encapsulation
- Separate implementation details from interface
  - Control access to internal data
    - Account class
  - Change class implementation without breaking code that uses the class
    - Point class

Access to Private Fields
- **Getter** and **Setter** methods
  - Point
    - `getX()`, `getY()`
    - `setX()`, `setY()`
- What not just make x, y public??

Package
- A collection of related classes and interfaces providing access protection and name space management
  - Group related classes together so they are easier to find and to use
  - Package level access finds a middle ground between public and private access
  - Avoid name conflicts
Creating Packages

```java
package cs202.cysun;
...
```

- Package names
  - The "reverse-URL" naming convention

Using Package Members

- Only public classes of a package are accessible from outside the package
- Import all classes in a package
  - E.g., `import javax.swing.*;`
- Import one class from a package
  - E.g., `import javax.swing.JOptionPane;`

Package and Directory

- Package name must match directory structure
  - E.g., all classes in the package `cs202.cysun` must be under a directory `cs202/cysun`
- Classpath – directories where Java searches for classes
  - Some default classpaths
  - Current working directory
  - Additional directories specified by the `-classpath` option

Package and Directory Example

```java
package cs202
- Class Foo

java Foo
java cs202.Foo
java -classpath .. cs202.Foo
```

Account Revisited

**Account**

- **Attributes**
  - Account number
  - Owner's name
  - Balance (≥ 0)

- **Operations**
  - Check balance
  - Deposit
  - Withdraw
  - Transfer

More Accounts

- **Checking Account**
  - No restriction on deposit or withdraw
- **Savings Account**
  - Limited 2 withdrawals per month
- **CD Account**
  - 30-day term
  - Deposit or withdraw only during a 7 day grace period
Inheritance

- Code re-use
- Subclass inherits members of a superclass
  - Class variables
  - Methods
  - Except constructors
- Inherits != Can Access
  - public and protected
  - Subclass may have more members than the superclass

CheckingAccount Class

```java
public class CheckingAccount extends Account {
    public CheckingAccount(String owner) {
        super(owner);
    }
    public CheckingAccount(String owner, double balance) {
        super(owner, balance);
    }
}
```

Keyword `super`

- A reference to the superclass
- A reference to a constructor of the superclass

SavingsAccount Class

- Restrictions on withdraw
  - No more than 2 withdraws per month
- Have to re-write the withdraw() method
  - ??

Overriding

- A subclass method has the same signature as a method of the superclass
- Method signature
  - Access modifier
  - Return Type
  - Name
  - List of parameters

Overriding Examples

- public double withdraw(double amount)
- public String toString()
  - All Java classes implicitly inherits from the Object class
  - toString() is one of the methods defined in the Object class
Inheritance vs. Encapsulation

- Inheritance – subclass wants to reuse code
- Encapsulation – changes to the implementation of one class should not affect other code, including the code of subclasses
- In practice – pragmatic balance

Class Hierarchy of Account

Account
- deposit()
- withdraw()
- transfer()

CheckingAccount
- withdraw()

SavingsAccount
- withdraw()

CDAccount
- deposit()
- withdraw()

Keyword final

- A final class cannot be inherited
  - public final class Classname {...}
- A final variable cannot change its value
  - Similar to constants in other languages
  - Convenience
  - Readability
  - final double PI = 3.1415926;

More about Account

Account

- Attributes
  - Account number
  - Owner’s name
  - Balance (>=0)

- Operations
  - Check balance
  - Deposit
  - Withdraw
  - Transfer

A Closer Look at transfer()

```java
public double transfer(double amount, Account other)
{
    return other.deposit(withdraw(amount));
}
```

- What happens if we want to transfer from a CheckingAccount to a SavingsAccount?
  - Type mismatch?

Things Could Get Messy

- CheckingAccount
  - double transfer(double amount, CheckingAccount a)
  - double transfer(double amount, SavingsAccount a)
  - double transfer(double amount, CDAccount a)

- SavingsAccount
  - double transfer(double amount, CheckingAccount a)
  - double transfer(double amount, SavingsAccount a)
  - double transfer(double amount, CDAccount a)

- CDAccount
  - double transfer(double amount, CheckingAccount a)
  - double transfer(double amount, SavingsAccount a)
  - double transfer(double amount, CDAccount a)
Polymorphism

◆ An object of a subclass can be used as an object of the superclass
  ◆ E.g. Account a = new CheckingAccount("Chengyu", 10.0);
◆ The reverse is not true
  ◆ E.g. CheckingAccount a = new Account("Chengyu", 10.0); // Error!
◆ Why??

Polymorphism Example

public class A {
    public A() {
    }
    public void afunc() {
        System.out.println(“afunc”);
    }
    public class B extends A {
        public B() {
        }
        public void bfunc() {
            System.out.println(“bfunc”);
        }
    }
    A a1 = new A();
    B b1 = new B();
    A a2 = new B(); // OK
    B b2 = new A(); // Error!
    a2.afunc(); // OK
    a2.bfunc(); // Error!
    ((B) a2).bfunc(); // OK
    ((B) a1).bfunc(); // Error!
}

Dynamic Dispatching

◆ When multiple implementations of the same method exist due to overriding, which method to invoke is determined by the actual class of the object
◆ Dynamic means the decision is made at runtime (as oppose to compile time)

Dynamic Dispatching Example

public class A {
    public A() {
    }
    public void afunc() {
        System.out.println(“afunc”);
    }
    A a = new A();
    B b = new B();
    A a2 = new B();
    a.afunc(); // ??
    b.afunc(); // ??
    a2.afunc(); // ??
}

public class B extends A {
    public B() {
    }
    public void bfunc() {
        System.out.println(“bfunc”);
    }
    public double transfer(double amount, Account other) {
        return other.deposit( withdraw(amount ) );
    }

◆ What happens if deposit() and/or withdraw() fails?